MINISTRY OF NATIONAL EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

Traffic Control System in the Connected Cars Context (TCS-3C)

LICENSE THESIS

Graduate: **Andrei TUDORICA**

Supervisor: **Prof. dr. eng. Rodica POTOLEA**

2019

MINISTRY OF NATIONAL EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

DEAN,	HEAD OF DEPARTMENT,
Prof. dr. eng. Liviu MICLEA	**Prof. dr. eng. Rodica POTOLEA**

Graduate: **Firstname LASTNAME**

LICENSE THESIS TITLE

1. **Project proposal:** *Short description of the license thesis and initial data*

2. **Project contents:** *(enumerate the main component parts) Presentation page, advisor's evaluation, title of chapter 1, title of chapter 2, …, title of chapter n, bibliography, appendices.*

3. **Place of documentation**: *Example*: Technical University of Cluj-Napoca, Computer Science Department

4. **Consultants**:

5. **Date of issue of the proposal:** November 1, 2018

6. **Date of delivery:** July 8, 2019

Graduate: ______________________________

Supervisor: ______________________________

MINISTRY OF NATIONAL EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

Declaraţie pe proprie răspundere privind
autenticitatea lucrării de licenţă

Subsemnatul(a)____
____,
legitimat(ă) cu _______________ seria ________ nr. ____________________________
CNP ____, autorul lucrării
____
____
____elaborată în vederea susţinerii examenului de finalizare a studiilor de licență la Facultatea de Automatică și Calculatoare, Specializarea ____ din cadrul Universităţii Tehnice din Cluj-Napoca, sesiunea __________________ a anului universitar ___________, declar pe proprie răspundere, că această lucrare este rezultatul propriei activităţi intelectuale, pe baza cercetărilor mele şi pe baza informaţiilor obţinute din surse care au fost citate, în textul lucrării, şi în bibliografie.

Declar, că această lucrare nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în faţa unei alte comisii de examen de licenţă.

In cazul constatării ulterioare a unor declaraţii false, voi suporta sancţiunile administrative, respectiv, *anularea examenului de licenţă*.

Data   Nume, Prenume

_______________________   __________________________________

Semnătura

Table of Contents

Chapter 1. Introduction

1.1. Traffic context

1.2. Connected Cars Context

Chapter 2. Project Objectives

2.1. Problem Overview

2.2. Project Objectives

The first objective of this project is to design and implement a system able to have a live overview of the Connected Cars Context. This system can be used in taking different actions related to traffic by several actors.

The second one is developing a simulator module in order to test the system. This simulator will mimic genuine traffic participants actions related to traffic. The system is will analyze and report the effect the traffic participants decisions have in the aforementioned context. Thereafter, this module allows the viewing of the traffic evolution under different parameters.

A list of objectives to be attained:

- Design a high-level architecture of the system in its basic form
- Decide on an open source map interpreter and route planning solution
- Setup the open source project for testing and development
- Understand all the features and limitations the open source project has
- Apply the open source solution to all the basic modules of the system
- Design a simulator module
- Decide on a set of cost parameters used for the study
- Decide on a list of metrics to study the simulated context
- Implement the simulator
- Compute and study the results of the simulations under different parameters

2.3. Requirements

2.3.1. Functional Requirements

Based on the presented objectives, the functional requirements resemble the features of the project. Each requirement is shortly described, but more details about the design and implementation of these features can be found in chapters 4 and 5.

- Component able to generate routes based on different factors
- Component able to hold the status of the traffic and update it when necessary
- Component able to simulate the traffic context
- Simulator should be able to display statistics over the length of the entire simulation
- Simulator should output detailed statistics at the end of the simulation

2.3.2. Nonfunctional Requirements

Chapter 3. Bibliographic Research

3.1. Navigation Systems

A navigation system is an electronic system that facilitates the navigation. They are composed of both software and hardware. Navigation systems can be offline, online or a combination of both.

Offline navigation systems assume that all the data is on board of the using party and doesn't need any kind of aid from remote components.

Online systems have components that are located in fixed points and the using party communicates with it in order to receive the navigation information needed.

The hybrid systems can have the map information stored locally but still request routes from a remote component with higher computational power.

3.1.1. Proprietary Navigation Systems (Closed Source)

A proprietary software is a kind of software that doesn't allow anybody but the developing team to see or modify the source code. This kind of software is usually developed by companies that sell that product or offer it for free but generate profit out of it.

The most used proprietary navigation systems are the ones that are offline. These come integrated with the car and are provided by the car manufacturer or can be purchased separately as GPS devices from separate manufacturers. These systems are able to provide routes, direction, traffic information (e.g. speed limits) even in the absence of an open internet connection.

On the other hand, with the development of the mobile devices, the online navigation systems got to increase their popularity and reach. By benefiting from internet connection these systems are able to offer the user a wider range of maps. They can also provide way more information on what concerns the current traffic status in certain areas, reviews and even street-level photos. The most relevant examples of online navigation apps are Google Maps and Waze. The fact online factor allows these apps to use user location in order to map the traffic and it allows them to use user data in order to show traffic information like traffic jams, police locations and many others. They offer the possibility to build a route in different ways: shortest distance, shortest time, least consumption, etc. Even if they have all these features, they still take into consideration only the user's choice when it comes to routing. They do not try to improve the entire context, but only the requesting users' route, even if they have one of the largest Connected Context when it comes to maps and navigation.

Considering the above mentioned, we are not able to use these systems directly in order to study and improve the Connected Cars Context.

3.1.2. Open Source Navigation Systems

An open source software is a type of software that are licensed by their developers in such a way that it grants users access to the code and it allows them to study, change and distribute the software, no matter the purpose. Most of these softwares are developed in a collaborative manner, many individual developers helping the process of designing, writing and testing the product.

These kinds of projects are the best choice for somebody that wants to try new routing techniques or even trying a new approach in the context of navigation systems. Writing a navigation software from scratch is a huge project and usually not worth it as the number of open source projects is big enough that it is almost sure that one can find a good starting point for their approach. Most of these projects are designed in such a way that they can be built over.

For this project, from the research of the open source community projects, two of them stood out.

- *OsmAnd*

  OsmAnd, presented in [1], is an open source software that runs on Android and iOS devices and is able to process OpenStreetMap data fully offline. OsmAnd has multiple navigation features like turn-by-turn voice guidance, traffic warnings (stop signs, pedestrian crosswalks, speed limits, etc.) and it has features for pedestrians and bicycle riders as well.

  The software was developed by a team of over 500 members and the entire source code is available on GitHub [2] under a license that allows any developer to modify and distribute the project, as long as they do it under the same type of license.

  The international community that takes part in the development of this product offered a huge collection of functionalities and elements on maps from over 40 countries, but it also brought a large set of undocumented features. The different coding styles approached by the developers affect the clarity of the projects code and makes it difficult for someone to build over it.

- *Itinero*

  Itinero, presented in [3], is an open source project built for route planning that was originally designed for logistical optimization. Just as OsmAnd [2], it processes OpenStreetMap data fully offline and it offers step by step instructions in routing. It can provide routes for cars, bicycles, pedestrians, mixed or custom profiles. On top of that, it can be used both as a library and a routing server for other projects, making it perfect for the development of other traffic related projects.

  The Itinero project was developed by Ben Abelshausen [4], who is a Belgian optimization algorithms and routing expert. The fact that this project is built by a single person brings ensures that the project is uniformly written and consistently documented.

3.2. Maps and Data

3.3. Routing

3.3.1. Graph forms

3.3.2. Algorithms

3.4. Communication Methods

Chapter 4. Analysis and Theoretical Foundation

4.1. Motivation for choosing Itinero

There are multiple reasons for which Itinero was chosen for developing this diploma project.

First of all, Itinero offers the possibility to be used as a library in any .NET project. That means the route computing is extremely customizable and can be used even for our very specific route computing criteria. It also means that the data structures used along the distributed components of this project are exactly the same, using as little effort to communicate as possible. The fact that Itinero is more of a tool than an application itself, allows the further development of a distributed system that uses it in all its components to be very easily extended.

Second of all, the way Itinero manipulates the map information is very well implemented, making it lightweight memory wise and fast even in the scenario of a distributed system, like in the case of this diploma project. Every data structure offers custom serialization and deserialization, which makes Itinero really fast when it comes to loading and transferring data.

Moreover, the fact that Itinero was developed by a single person (with confirmed design, algorithmic and development skills) ensures that the project is uniformly written. The code flows are consistent and well implemented and coding style is the same in the entire project, so the entire software is comfortable to study and understand, even if it is really heavy when it comes to the content of the code.

To complete the above stated argument, Itinero comes with a good documentation, both in the wiki page and in code comments. Because of this, understanding it and finding answers to almost any technical questions regarding Itinero is not as difficult as it might be in the case of other projects. Besides that, as this diploma project required some really specific tools from Itinero's kit, whenever something was not clear enough, Ben, the developer of Itinero, was kind enough to provide the needed help.

Last but not least, the technical stack used in the development of Itinero, .NET framework written in C#, completed the list of arguments for taking this decision.

4.2. Conceptual Architecture of the System

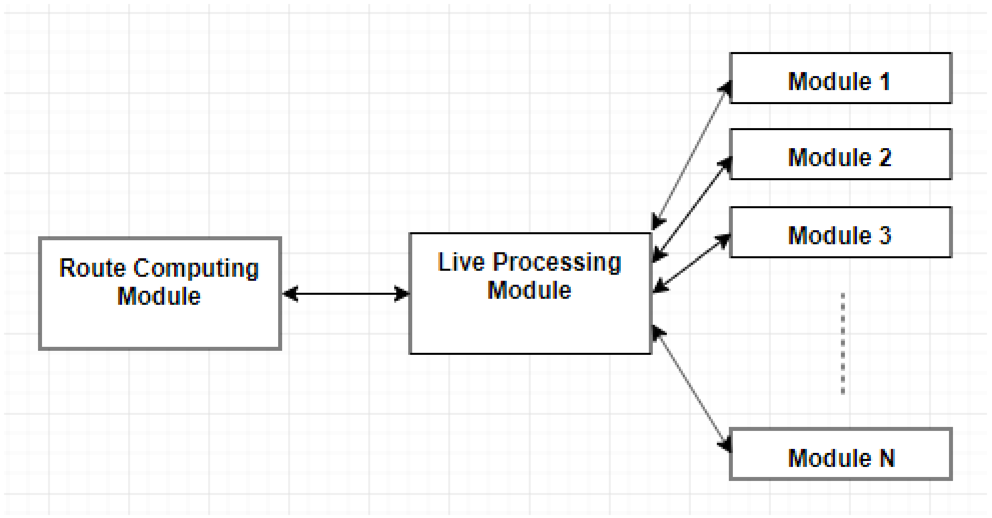


Figure 4.1 Conceptual Architecture of the TCS-3C

As a distributed system, the currently described project is built using modules. The main components are the Live Processing Module and the Route Computing Module. All other possible modules would communicate directly to the Live Processing Module.

4.2.1. Live Processing Module

The Live Processing Module is the core of this system. It is the module that supports or controls the actions of every other component in the system. It also holds the current state of the traffic.

This module has 2 types of input: route requests and notifications relating traffic. These inputs are received from Active Modules (4.3.2.). This module can also request traffic information from Passive Modules, in order to keep the traffic status, it holds up to date.

When a route request is received, the Live Processing Module requests the latest traffic data from the available Passive Modules (4.3.2.), in order to ensure the reliability of the currently held traffic data. Then it makes a request to the Route Computing Server for a route and after it receives the route, it sends it to the requesting module.

4.2.2. Route Computing Module

The Route Computing Module is the component of the system that takes care exclusively of computing routes, considering only the c2c infrastructure. It receives as input a route request and it returns the computed route based on the current status of the traffic. Whenever a route request is received, this module loads the traffic data that was updated by the Live Processing module.

4.2.3. Modules

- Active Modules

Active Modules are the type of modules that take direct actions in or regarding the traffic context. These modules are usually Client applications for Traffic Researchers, Traffic Participants or Traffic Managers. These modules are the ones that provide the input to the Live Processing Module and they process the responses received.

- Passive Modules

Passive Modules are the type of modules that hold a traffic status of their own based on their own input and receive requests from the Live Processing Module. They provided their current status to the Live Processing Server and it will take this information into consideration when updating the general traffic status.

4.2.4. Example of System build on the Conceptual architecture

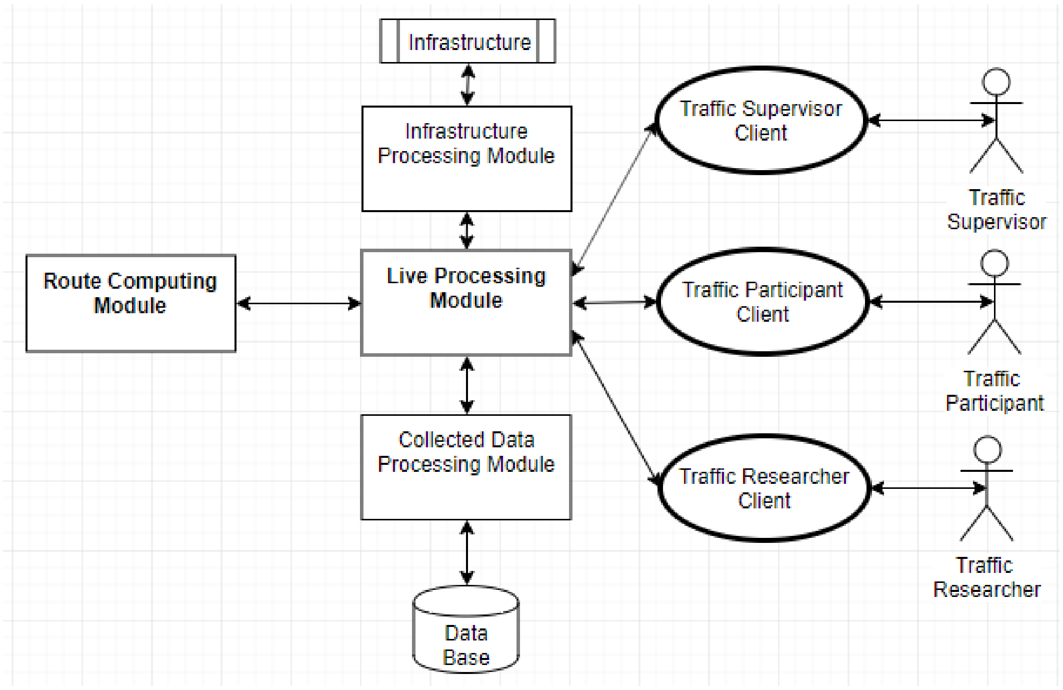

Infrastructure
Infrastructure Processing Module
Route Computing Module
Live Processing Module
Collected Data Processing Module
Data Base
Traffic Supervisor Client
Traffic Participant Client
Traffic Researcher Client
Traffic Supervisor
Traffic Participant
Traffic Researcher

Figure 4.2 Example of architecture based on the conceptual architecture

Figure 4.2. describes a system built on the conceptual architecture presented in the previous pages (4.3). The system above respects the structure proposed and has the main components and 5 other modules (3 active modules and 2 passive modules).

- **Live Processing Module**

As described at point 4.2.1., the Live Processing Module is the core module of the system. It holds the current traffic status, updates it constantly and communicates with all the other modules of the system.

The Live Processing Module has 2 type of inputs. The first type of input is traffic status updates, received from the Traffic Participant Client Active Module, Traffic Manager Client Active and, on request, from the 2 Passive Modules of the system. The Live Processing Module uses these updates to keep the traffic status up to date.

The second type of input received are requests. The Traffic Participant Client can request routes and the Live Processing Server forwards the request to the Route Computing Module and after it receives the response it outputs that response to the Traffic Participant client. On the other hand, the Traffic Researcher Client can request for statistics on different topics of the traffic status. When that input is received, the Live Processing Server computes and outputs the requested statistics.

- **Route Computing Module**

  As described at point 4.2.2., the Route Computing Module has the purpose to answer route requests with a route, based on the current status of the traffic, if it is possible to build one, or with an error message if the route cannot be computed.

- **Active Modules**

- **Traffic Participant Client**

  The Traffic Participant Client Module is the client application that can be used by the Traffic Participant stakeholder. This module aids the Traffic Participant in the process of requesting a route.

  After a route is received the module presents the Traffic Participant with the route, guides them and sends constant location updates to the Live Processing Module, so the traffic status can be updated.

- **Traffic Manager Client**

  The Traffic Manager Client Module is the client application that can be used by the Traffic Manager stakeholder. It offers the Traffic Manager the possibility to bring changes to the traffic context, like blocking a certain area of the city for various events.

- **Traffic Researcher Client**

  The Traffic Researcher Client Module is the client application that can be used by the Traffic Researcher stakeholder. It offers the Traffic Researcher the possibility to request the Live Processing Module for various statistics on what concerns the traffic.

- **Passive Modules**

- **Collected Data Processing Module**

  The Collected Data Processing Module is the passive module that is responsible with storing the traffic data received or computed by the Live Traffic Server. This module has direct connection to a database in which it stores the collected data, and from which it retrieves and analyses traffic data. It can provide the Live Traffic Module, on demand, with information regarding the overall usual status of the traffic at certain times/dates in the past.

- **Infrastructure Processing Module**

  The Infrastructure Processing Module is the passive module that is responsible for communicating with the infrastructure systems, like live traffic cameras or proximity car counters, and build a status traffic model based on the information received from these systems. It is able to answer the Live Traffic Modules requests with the live status traffic computed using the traffic data from the infrastructure.

- A route request Use-Case, from the perspective of the Live Traffic Module, is presented in the following flow diagram.

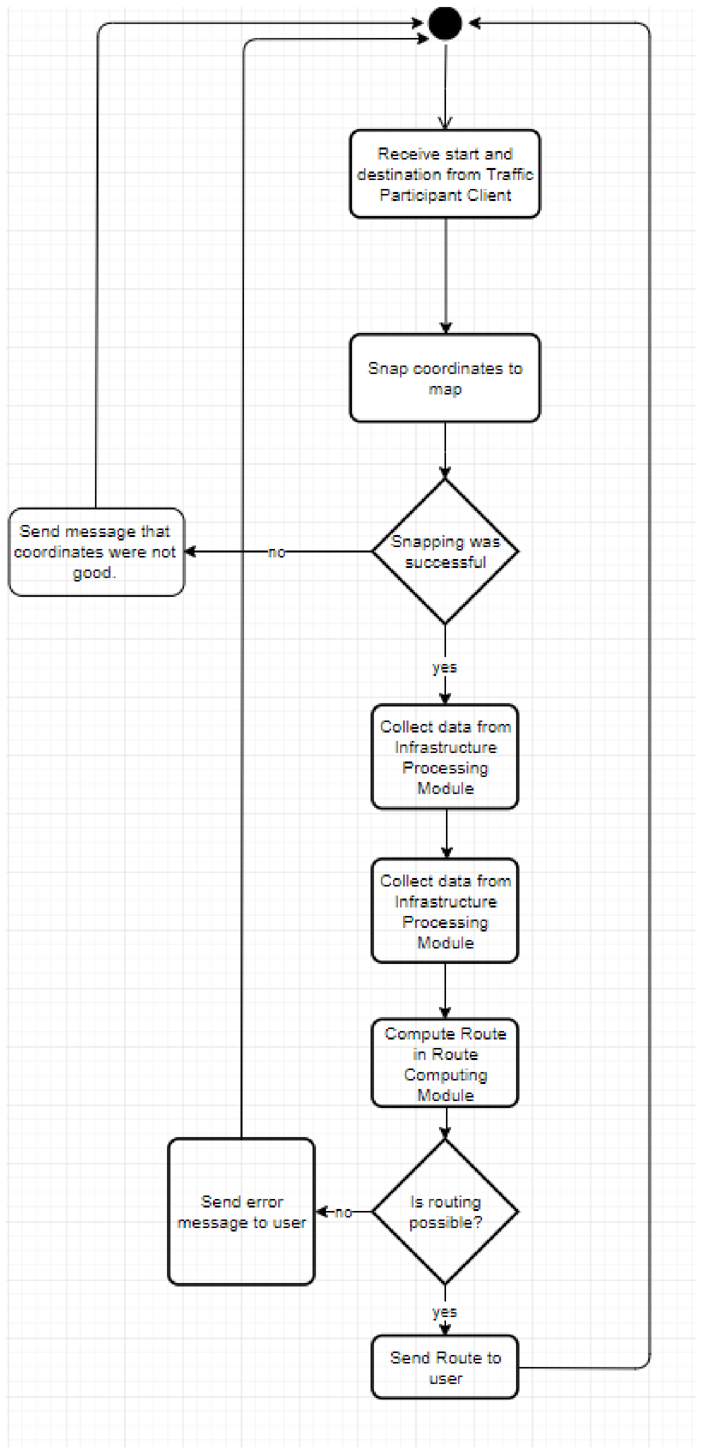

Receive start and destination from Traffic Participant Client
Snap coordinates to map
Snapping was successful
no
Send message that coordinates were not good.
yes
Collect data from Infrastructure Processing Module
Collect data from Infrastructure Processing Module
Compute Route in Route Computing Module
Is routing possible?
no
Send error message to user
yes
Send Route to user

Figure 4.3 Flow diagram of route request Use-Case

4.3. TCS-3C - Design

4.3.1. Design

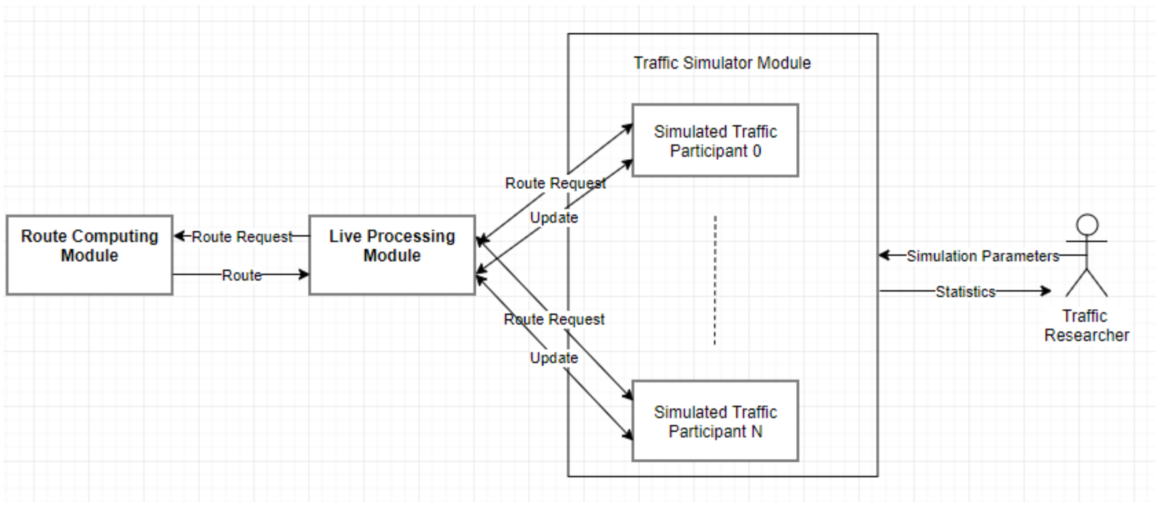


Figure 4.4 Design of TCS-3C

Figure 4.4. describes the architecture of TCS-3C. It respects the conceptual diagram described earlier (4.2.). It has the main modules, the Live Traffic Module and the Route Computing Module, and one active module, the Traffic Simulator Module.

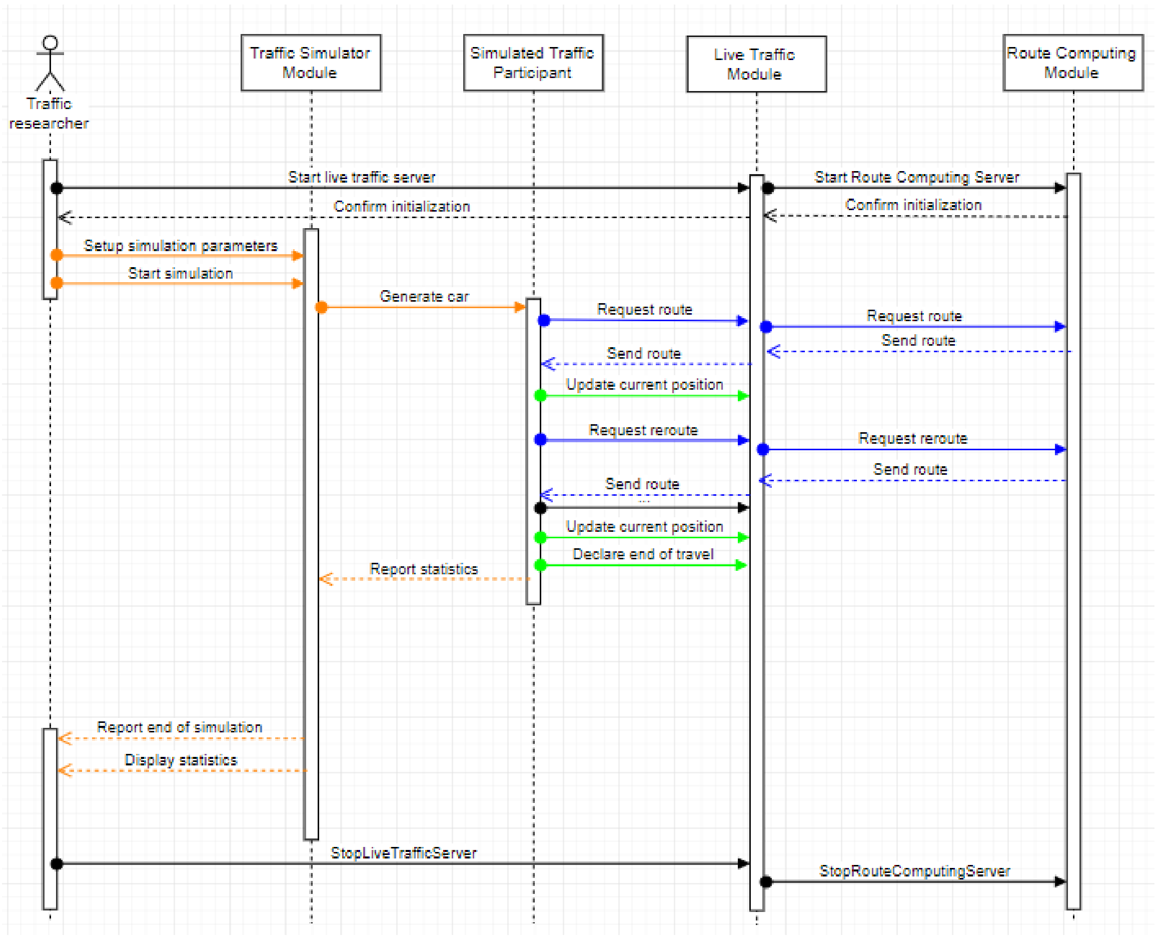


Figure 4.5 Sequence Diagram of a TCS-3C

Figure 4.5. describes a simulation in terms of inter-module communication in the TCS-3C system.

- **Live Processing Module**

As it was previously described, The Live Processing Module is the main module of the system. It has the purpose of connecting all other modules in the system. In this case, the Live Processing Module has a direct connection to the Route Computing Module and to the Traffic Simulator Module.

This module manages the request for routes received from the Simulator Module. Whenever a route request is received, the Live Traffic Module forwards it to the Route Computing Module, together with the current status of the traffic. It then expects a route and forwards it to the simulated traffic participant. These actions can be found colored blue in Fig 4.4.

It maintains the current status of the traffic by updating it periodically when location updates are received from traffic participants simulated by the Traffic simulator. These actions can be seen in color Green in figure 4.4.

- **Route Computing Module**

As described at point 4.2.2., the Route Computing Module has the purpose to answer route requests with a route, based on the current status of the traffic, if it is possible to build one, or with an error message if the route cannot be computed.

- **Traffic Simulator Module**

This module is the interface of the Traffic Researcher in TCS-3C. Its purpose is to generate a traffic context based on a set of parameters and create statistics on the effects of Traffic Participants choices, made in traffic. These effects can be studied by varying a threshold (0%-100%) of people taking the best route over the shortest one and comparing the total times of the simulated contexts.

It simulates traffic participants, giving each one of them a type of behavior, a starting location and an ending location. Each one of these traffic participants requests a route and starts following it, according to their behavior, updating their position and requesting reroutes when necessary.

During the simulation process, the module gathers statistics from the simulated traffic participants and displays them to the user at different points in the simulation and at the end of it.

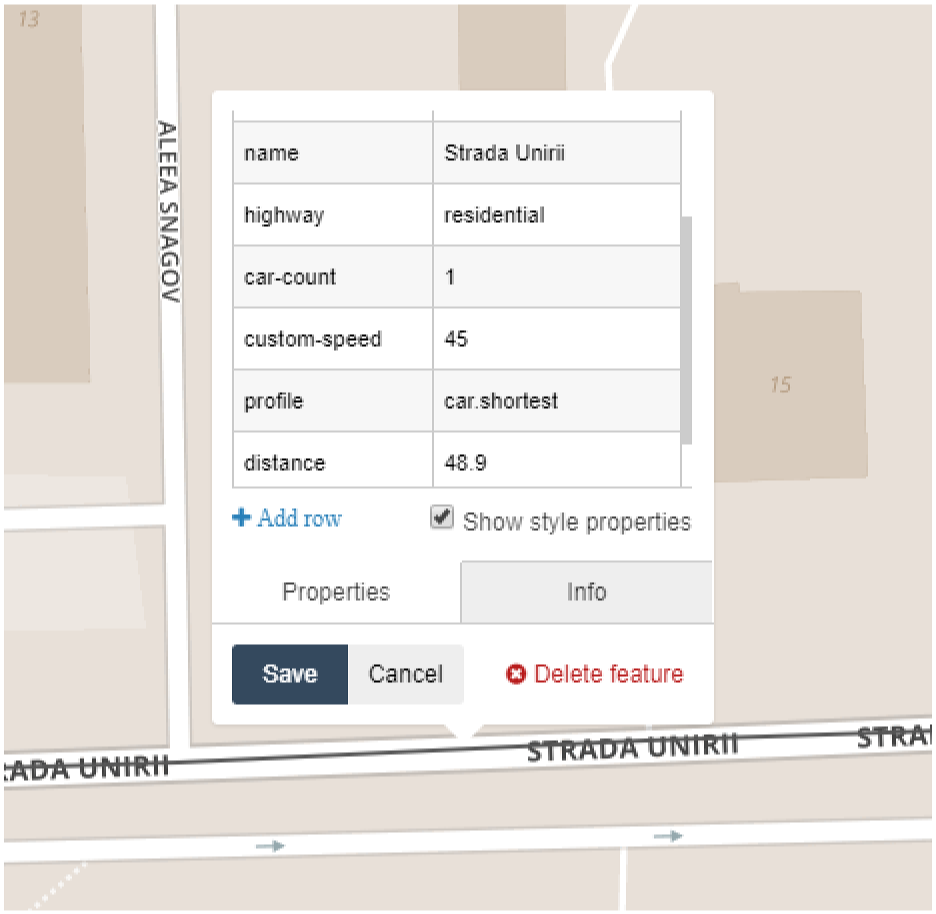
These actions can be seen in Fig 4.4. colored orange.

4.3.2. Traffic Data

Previous points stated that the Live Traffic Module holds the traffic status in order to be used when the routing is done. This traffic status is a combination of the already existent map information and the traffic data collected by the Live Traffic Module. Besides the basic data related to the maps, presented at 3.x(TAG), in order to hold the traffic related data, each edge has two additional characteristics:

- car-count: The number of cars that can be found on that edge at the time of the query
- custom-speed: A speed that was computed for that edge

An example can be found in the following image.



name	Strada Unirii
highway	residential
car-count	1
custom-speed	45
profile	car.shortest
distance	48.9

Figure 4.6 Screenshot of partial information recorded about an edge on the map

Normally, the speed associated with an edge is hardcoded and is equal to the speed limit legally imposed for that edge. In the case of TCS-3C, besides the legal speed limit, the number of cars present at one time on that edge and their average length influences the speed that can be reached on that edge. The function that determines the new speed can be easily changed in the system and the current form of this function is the following:

$$custom\_speed = legal\_speed * (1 - \frac{car\_count * average\_car\_length}{edge\_distance})$$

For example, in the picture above the legal speed limit is 50 km/h in that area, the number of cars on that edge is 1, the average car length is 4 meters and the edge distance is 48.9 meters. That gives us, using the formula above, a custom speed of 45.7 km/h, which is rounded to 45.

This function dictates the new speed for every edge that will be used for routing.

4.3.3. Routing process

The routing process takes place in the Route Computing Module.

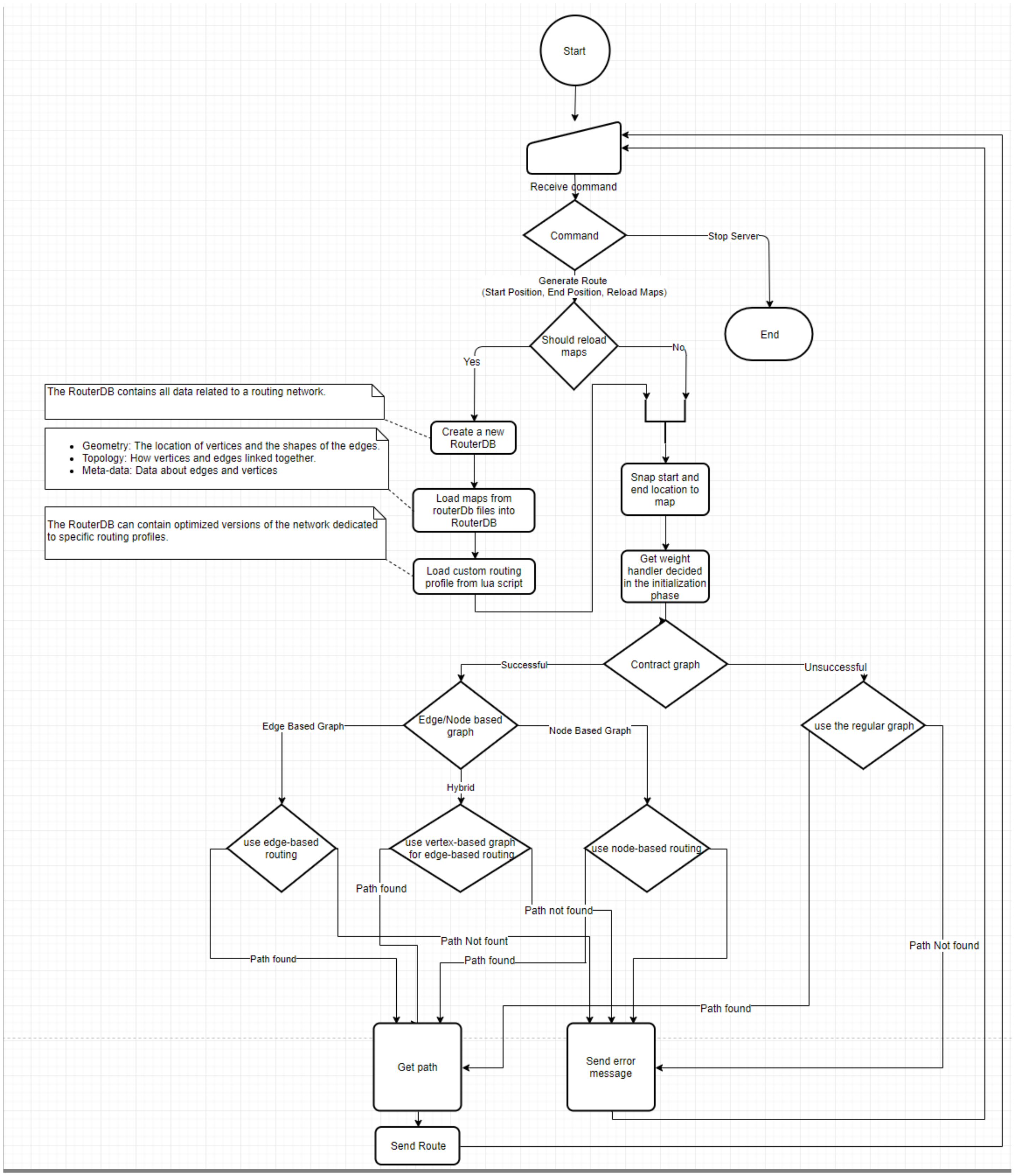

Start
Receive command
Command
Stop Server
End
Generate Route
(Start Position, End Position, Reload Maps)
Should reload maps
Yes
No
The RouterDB contains all data related to a routing network.
Create a new RouterDB
• Geometry: The location of vertices and the shapes of the edges.
• Topology: How vertices and edges linked together.
• Meta-data: Data about edges and vertices
Load maps from routerDb files into RouterDB
The RouterDB can contain optimized versions of the network dedicated to specific routing profiles.
Load custom routing profile from lua script
Snap start and end location to map
Get weight handler decided in the initialization phase
Contract graph
Successful
Unsuccessful
Edge/Node based graph
Edge Based Graph
Node Based Graph
Hybrid
use the regular graph
use edge-based routing
use vertex-based graph for edge-based routing
use node-based routing
Path found
Path not found
Path Not fount
Path found
Path found
Path found
Path Not found
Get path
Send error message
Send Route

Figure 4.7 Flow Chart of route computing process

The first steps of the process include deciding whether the traffic status should be reloaded or not. If there were enough changes in the traffic context, the routerDb is reloaded.

The second part of the routing process is validating the start and end locations. It consists of checks whether the two pairs of coordinates are included in the map that is currently concerned. If the points are inside the map, the next step is checking if these points can be snapped to the road network in the map or if they are too far.

After the input data is validated, the next step is trying to contract the graph. This step minimizes the size of the graph so less effort is required in the route computing process. This process removes from the routerDb the information that is not used for the current routing profile. For example, if the current routing is only done for cars, all information about pedestrians and bicycles can be removed from the routerDb, as it won't be used. This step also removes the nodes that have under three neighbors. The contraction also includes transforming the basic graph into an edge-based graph, if possible. This action would make the edges the main point of interest in the routerDb and the nodes would become edges. This would allow speed up in the routing process and would make it easier to model turn restrictions.

The last part of the process is the actual routing. Route computing is done using a custom bidirectional Dijkstra's algorithm, for each form of the graph. This algorithm There are 2 types of routing in TCS-3C:

- shortest - a bidirectional Dijkstra's algorithm adaptation that takes as weights the length of the edges and the legal speed limit. This type of routing is used by the greedy Traffic Participants.
- best - a bidirectional Dijkstra's algorithm adaptation that takes as weights the length of the edges and the custom speed computed for each edge (4.3.2)

Chapter 5. Detailed Design and Implementation

5.1. Design

In terms of implementation, the TCS-3C is built using Microsoft's technical stack. The Live Traffic Server and the Route Computing Server are both .NET Core 2.1 APIs, while the Traffic Simulator is a Console Application in .NET Framework 4.6.1. In this chapter each one of these elements will be discussed in terms of implementation and the communication between these modules.

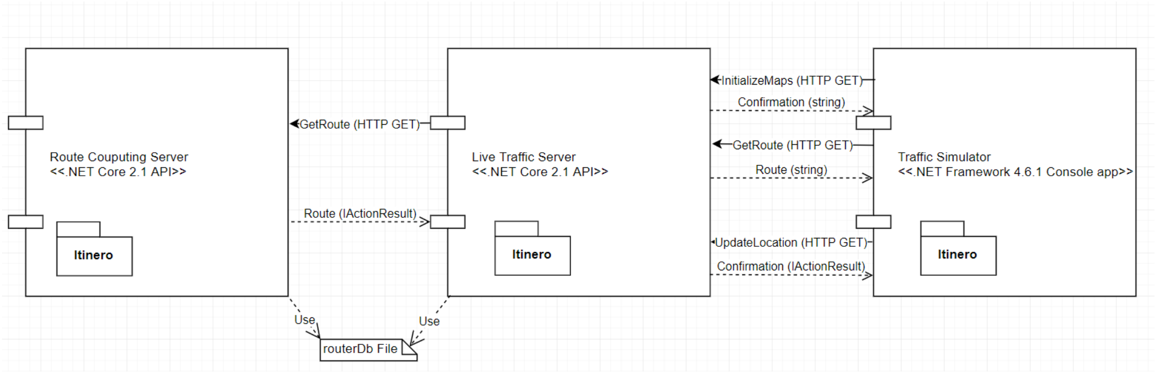


Figure 5.1 High Level design of TCS-3C

In Figure 5.1, a high level design of the system is described. At first sight, it is visible that all the modules in the system have the Itinero package. This package helps keep the datastructures similar over the entire system.

In terms of communication, as discussed at point 3.4, the TCS-3C presents an inter module communication based on the HTTPS protocol. Each one of the main components exposes several endpoints that are accessed via HTTP requests. The only exception concerning the communication is the way the Live Traffic Server communicates the map status: it is done using the routerDb file.

5.2. Live Traffic Server

5.2.1. Design

The Live Traffic Server is a .NET Core 2.1 API that, as described at point 4.3 is the core server of the application.

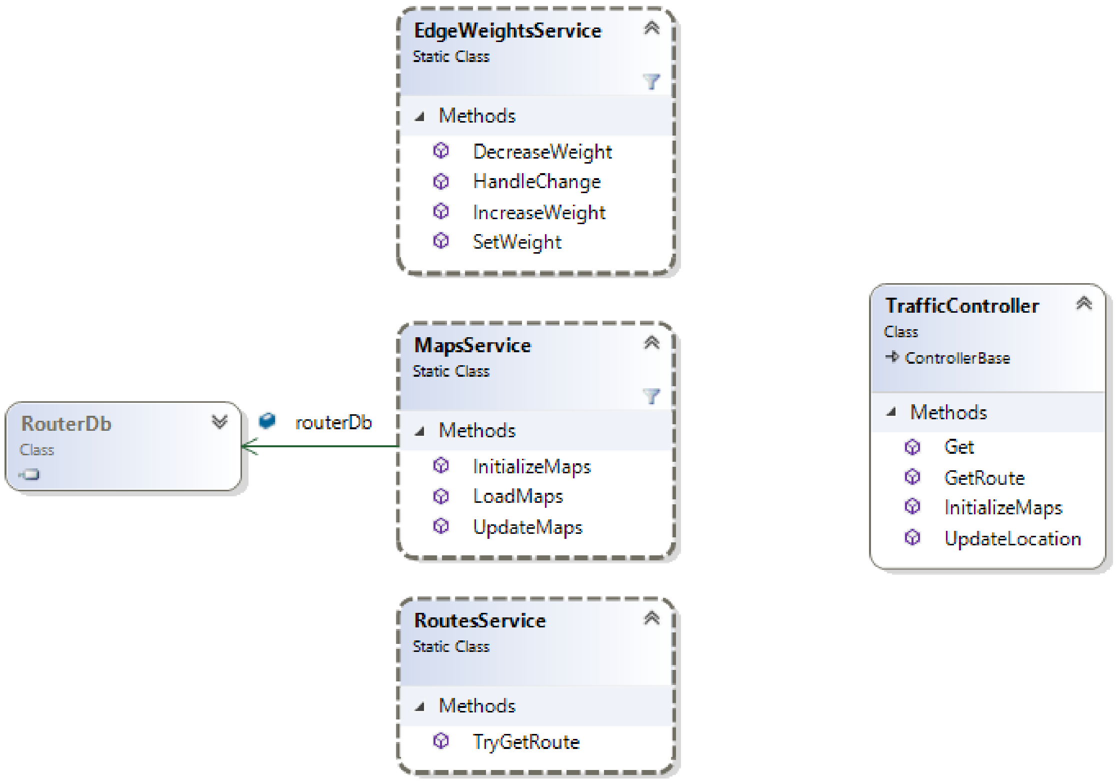
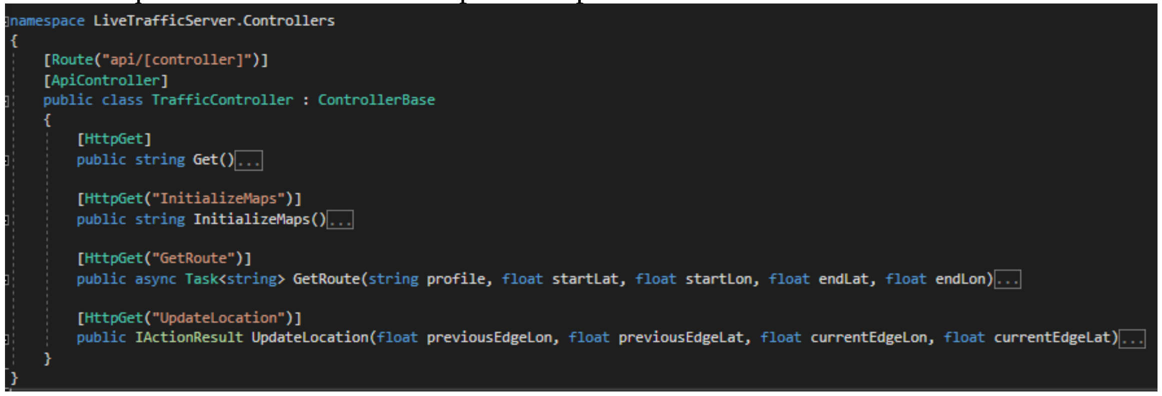


Figure 5.2 High level class diagram of Live Traffic Server (from Visual Studio)

Figure 5.2 presents the high level class diagram of the Live Traffic Server. For the current version of TCS-3C the high level architecture is rather simple. It contains four main classes:

Traffic Controller: is an extension of the ControllerBase class in .NET Core. It is responsible with exposing the endpoints and handling the requests. There are four endpoints exposed in the Traffic Controller:



```
namespace LiveTrafficServer.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class TrafficController : ControllerBase
    {
        [HttpGet]
        public string Get()...

        [HttpGet("InitializeMaps")]
        public string InitializeMaps()...

        [HttpGet("GetRoute")]
        public async Task<string> GetRoute(string profile, float startLat, float startLon, float endLat, float endLon)...

        [HttpGet("UpdateLocation")]
        public IActionResult UpdateLocation(float previousEdgeLon, float previousEdgeLat, float currentEdgeLon, float currentEdgeLat)...
    }
}
```

Figure 5.3 Screenshot of the Traffic Controller code, methods colapsed to deffinitions

Figure 5.3 presents the endpoints of the Traffic Controller.

The first one is a test GET endpoint that returns „Live Traffic Server is Running. Waiting for commands.” if the server is running.

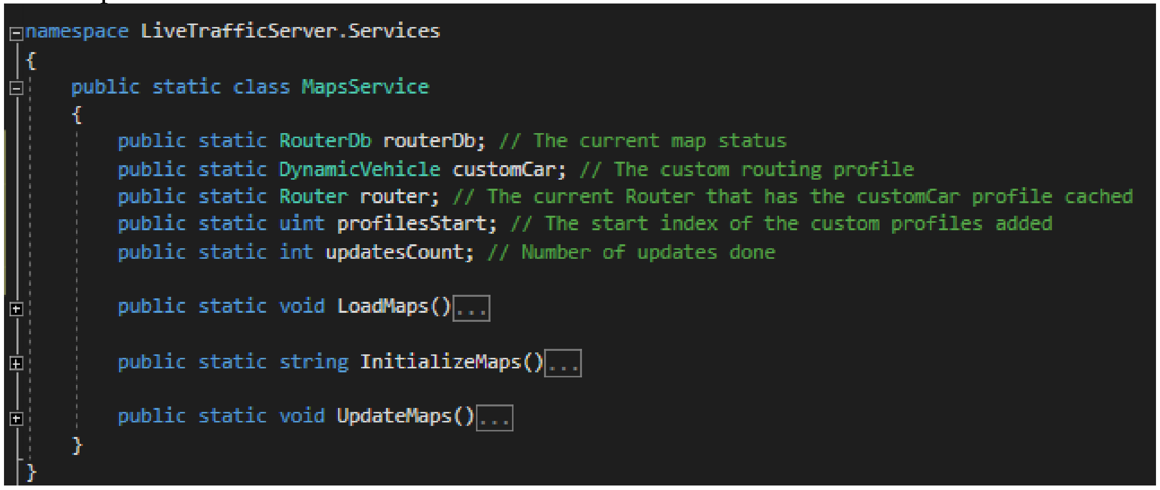
The second one is the InitializeMaps GET endpoint which calls the MapsService.InitializeMaps() method. It returns a confirmation string.

The third one is the GetRoute endpoint which receives five parameters: the routing profile (shortest or best), latitude and longitude of the start and end location. This endpoint delivers the resonsability to the RoutesService by calling the TryGetRoute method in the service with the received parameters. It returns the route received from the RoutesService as a string.

The fourth endpoint is UpdateLocation. It receives as parameters the coordinates of the middle of the edge a Traffic Participant is currently leaving and the coordinates of the middle of the edge the participant is moving to. It calls the EdgeWeightsService.HandleChange method with the received parameters. It returns an OK(„succesful") ActionResult if the update was succesful of a BadRequest(errorMessage) otherwise.

- **MapsService**

The Maps Service is a static service that is responsible with maintaining the current status of the maps and handling all the actions performed with it.



```
namespace LiveTrafficServer.Services
{
    public static class MapsService
    {
        public static RouterDb routerDb; // The current map status
        public static DynamicVehicle customCar; // The custom routing profile
        public static Router router; // The current Router that has the customCar profile cached
        public static uint profilesStart; // The start index of the custom profiles added
        public static int updatesCount; // Number of updates done

        public static void LoadMaps()...

        public static string InitializeMaps()...

        public static void UpdateMaps()...
    }
}
```

Figure 5.4 Screenshot of the MapsService code, methods colapsed to definitions

Figure 5.4 presents the main elements of the MapsService. This service has five static members and three static methods.

The first static member is the routerDb. This object holds the entire map and profiles associated with each edge. It is initialized in the LoadMaps or in the InitializeMaps method.

The second static member is the customCar DynamicVehicle. It is used for snapping points to the map and contracting of the graph (4.3.3). This is loaded in the InitializeMaps method from the custom .lua script.

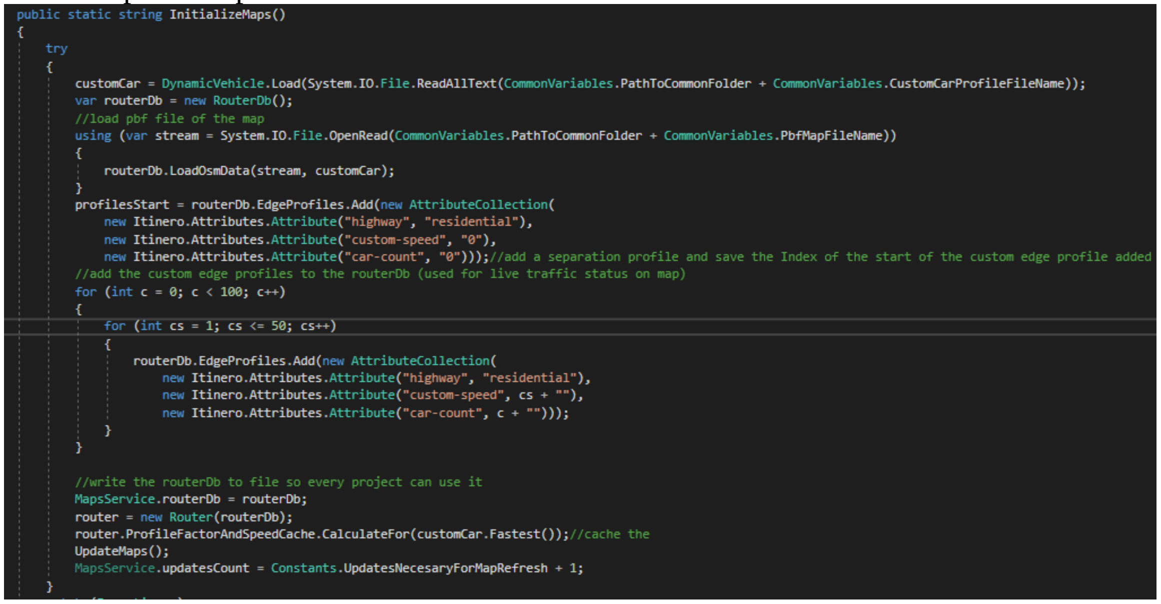
The third static member is the router. This member chaches the costomCar profile and it is used for snaping points to the map.

The profilesStart static member holds the index where the custom edge profiles start.

The updateCount static member holds the number of updates done. This member is used for the Live Traffic server to know when to Update the maps in the routerDb file and flag the Route Computing server that it should reload the maps aswell. The maps are updated and this member resets when the UpdatesNecesaryForMapRefresh parameter in Constants.cs is smaller than the number of updates.

The first method is LoadMaps. It is used to load the routerDb from a file.

The second static method is the InitializeMaps method. It loads a new routerDb directly from the pbf format of the map. On the loaded map it adds 500 new custom edge profiles. One for each combination of car-count (0-99) and custom-speed (1-50). It initializes the customCar routing profile, caches it into the router and updates the routerDb file, by calling the UpdateMaps method.



```
public static string InitializeMaps()
{
    try
    {
        customCar = DynamicVehicle.Load(System.IO.File.ReadAllText(CommonVariables.PathToCommonFolder + CommonVariables.CustomCarProfileFileName));
        var routerDb = new RouterDb();
        //load pbf file of the map
        using (var stream = System.IO.File.OpenRead(CommonVariables.PathToCommonFolder + CommonVariables.PbfMapFileName))
        {
            routerDb.LoadOsmData(stream, customCar);
        }
        profilesStart = routerDb.EdgeProfiles.Add(new AttributeCollection(
            new Itinero.Attributes.Attribute("highway", "residential"),
            new Itinero.Attributes.Attribute("custom-speed", "0"),
            new Itinero.Attributes.Attribute("car-count", "0")));//add a separation profile and save the Index of the start of the custom edge profile added
        //add the custom edge profiles to the routerDb (used for live traffic status on map)
        for (int c = 0; c < 100; c++)
        {
            for (int cs = 1; cs <= 50; cs++)
            {
                routerDb.EdgeProfiles.Add(new AttributeCollection(
                    new Itinero.Attributes.Attribute("highway", "residential"),
                    new Itinero.Attributes.Attribute("custom-speed", cs + ""),
                    new Itinero.Attributes.Attribute("car-count", c + "")));
            }
        }

        //write the routerDb to file so every project can use it
        MapsService.routerDb = routerDb;
        router = new Router(routerDb);
        router.ProfileFactorAndSpeedCache.CalculateFor(customCar.Fastest());//cache the
        UpdateMaps();
        MapsService.updatesCount = Constants.UpdatesNecesaryForMapRefresh + 1;
    }
```

Figure 5.5 Screenshot of InitializeMaps method of the MapsService

Figure 5.5 presents the code described in the previos paragraphs.

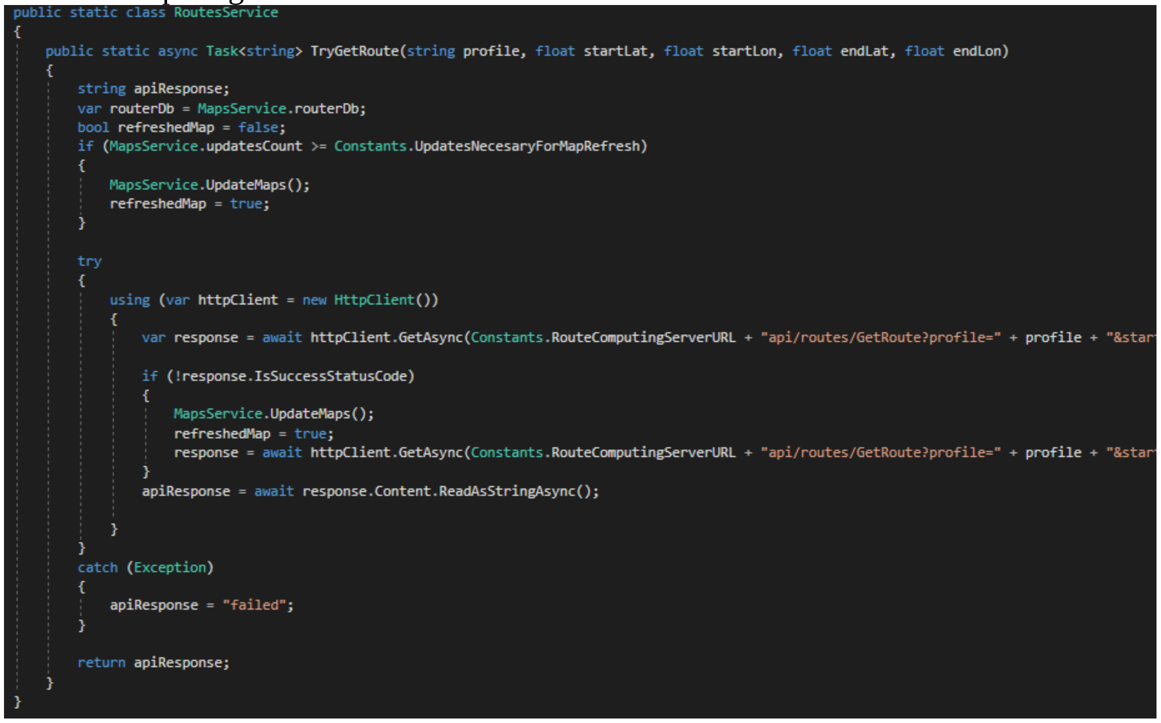
The last static method is the UpdateMaps. It is used to serialize the routerDb and write it into the routerDb file.

- **RoutesService**

  This static class is has the responsability to handle the routing requests. It has only one static asyncronous method TryGetRoute. This method receives as parameters form the TrafficController a routing profile name („shortest" or „best") and the coordinates of the start and end point of the request.

  Firstly, if the updatesCount in the MapsService is greater or equal than the UpdatesNecesaryForMapRefresh parameter in Constants.cs, The Routes Service calls the UpdateMaps method of the MapsService and flags that the following route request to the Route Computing Server should notify the changing of the routerDb file.

Then, it builds a HTTP request and calls the Route Computing Server for a route. In the case of a BadRequest it updates the maps again and tries a new call by flagging the Route Computing Server to refresh the maps it has. The last step is returning the response received from the Route Computing Server to the TrafficContriller. The code can be seen below.

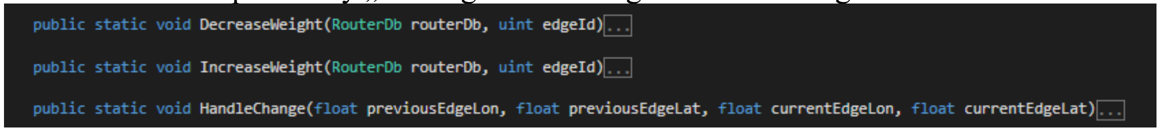


```
public static class RoutesService
{
    public static async Task<string> TryGetRoute(string profile, float startLat, float startLon, float endLat, float endLon)
    {
        string apiResponse;
        var routerDb = MapsService.routerDb;
        bool refreshedMap = false;
        if (MapsService.updatesCount >= Constants.UpdatesNecesaryForMapRefresh)
        {
            MapsService.UpdateMaps();
            refreshedMap = true;
        }

        try
        {
            using (var httpClient = new HttpClient())
            {
                var response = await httpClient.GetAsync(Constants.RouteComputingServerURL + "api/routes/GetRoute?profile=" + profile + "&star

                if (!response.IsSuccessStatusCode)
                {
                    MapsService.UpdateMaps();
                    refreshedMap = true;
                    response = await httpClient.GetAsync(Constants.RouteComputingServerURL + "api/routes/GetRoute?profile=" + profile + "&star
                }
                apiResponse = await response.Content.ReadAsStringAsync();

            }
        }
        catch (Exception)
        {
            apiResponse = "failed";
        }

        return apiResponse;
    }
}
```

Figure 5.6 Screenshot of the RoutesService code

- **EdgeWeightsService**

  The EdgeWeightsService static class is responsible with managing all the actions taken in the map at edge level. This is where the status of the traffic is updated by „moving” cars through the virtual edges.

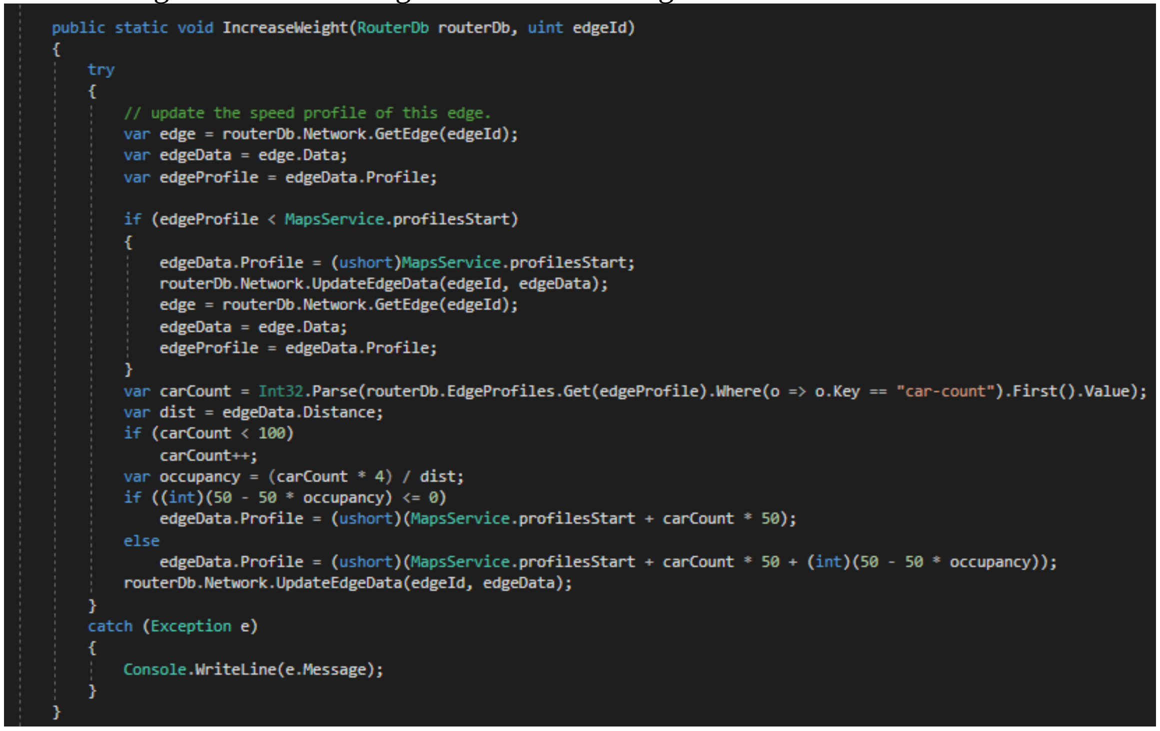


```
public static void DecreaseWeight(RouterDb routerDb, uint edgeId)...

public static void IncreaseWeight(RouterDb routerDb, uint edgeId)...

public static void HandleChange(float previousEdgeLon, float previousEdgeLat, float currentEdgeLon, float currentEdgeLat)...
```

Figure 5.7 Screenshot of EdgeWeightsService methods colapsed to definitions

Figure 5.7 presents the three static methods of the EdgeWeightsService. A change in the traffic context is the moment when a Traffic Participant goes from one edge to another. To deonte that, the TrafficControllers’ UpdateLocation endpoint is called with two coordinates: the middle of the edge the Traffic Participant is leaving and the middle of the edge the Traffic Participant is joining. The TrafficController calls the HandleChange static method of the EdgeWeightsService.

The HandleChange static method takes as arguments two sets of coordinates (previousEdgeLon, previousEdgeLat) and (currentEdgeLon, currentEdgeLat). If the previous coordinate is (0,0) that means a new traffic participant is joining the traffic. If the previous coordinate is not (0,0) but the current one is, that means a traffic participant just left the traffic. If none of the 2 coordinates are (0,0) that denotes only the action of an already existing traffic participant. The HandleChange method snaps the nonzero coordinates to the map and gets the Id of the edges taken into consideration. The next step is decreasing the weight of the previous weight, as a car just left that edge, and increasing the weight of the current edge. This is done using the DecreaseWeight and IncreaseWeight static methods.



```
public static void IncreaseWeight(RouterDb routerDb, uint edgeId)
{
    try
    {
        // update the speed profile of this edge.
        var edge = routerDb.Network.GetEdge(edgeId);
        var edgeData = edge.Data;
        var edgeProfile = edgeData.Profile;

        if (edgeProfile < MapsService.profilesStart)
        {
            edgeData.Profile = (ushort)MapsService.profilesStart;
            routerDb.Network.UpdateEdgeData(edgeId, edgeData);
            edge = routerDb.Network.GetEdge(edgeId);
            edgeData = edge.Data;
            edgeProfile = edgeData.Profile;
        }
        var carCount = Int32.Parse(routerDb.EdgeProfiles.Get(edgeProfile).Where(o => o.Key == "car-count").First().Value);
        var dist = edgeData.Distance;
        if (carCount < 100)
            carCount++;
        var occupancy = (carCount * 4) / dist;
        if ((int)(50 - 50 * occupancy) <= 0)
            edgeData.Profile = (ushort)(MapsService.profilesStart + carCount * 50);
        else
            edgeData.Profile = (ushort)(MapsService.profilesStart + carCount * 50 + (int)(50 - 50 * occupancy));
        routerDb.Network.UpdateEdgeData(edgeId, edgeData);
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

Figure 5.8 Screenshot of the IncreaseWeight method of the EdgeWeights static class

The IncreaseWeight and DecreaseWeight work in a similar manner. They receive as parameters the routerDb and an edge Id. First they retreive the profile of the edge. If this profiles Id is lower than the profilesStart variable in the MapsService it means that is hasn't been visited before and it is assigned the first custom edge profile (car-count =0, custom-speed=0). The next step is to compute the new number of cars, by retreiving the car-count value and increasing/decreasing it. By aplying the formula described at 4.3.2, a new custom-speed is computed. Using the new car-count and custom-speed, the Id of the new edge profile is computed and assigned to the edge.

5.2.2. Flow

The live Traffic Server has three main flows, associated with the endpoints exposed by the Traffic Controller: InitializeMaps, UpdateLocation and GetRoute. These flows take different code paths through the previously described Classes and methods in order to return an answer.

- **The InitializeMaps flow**

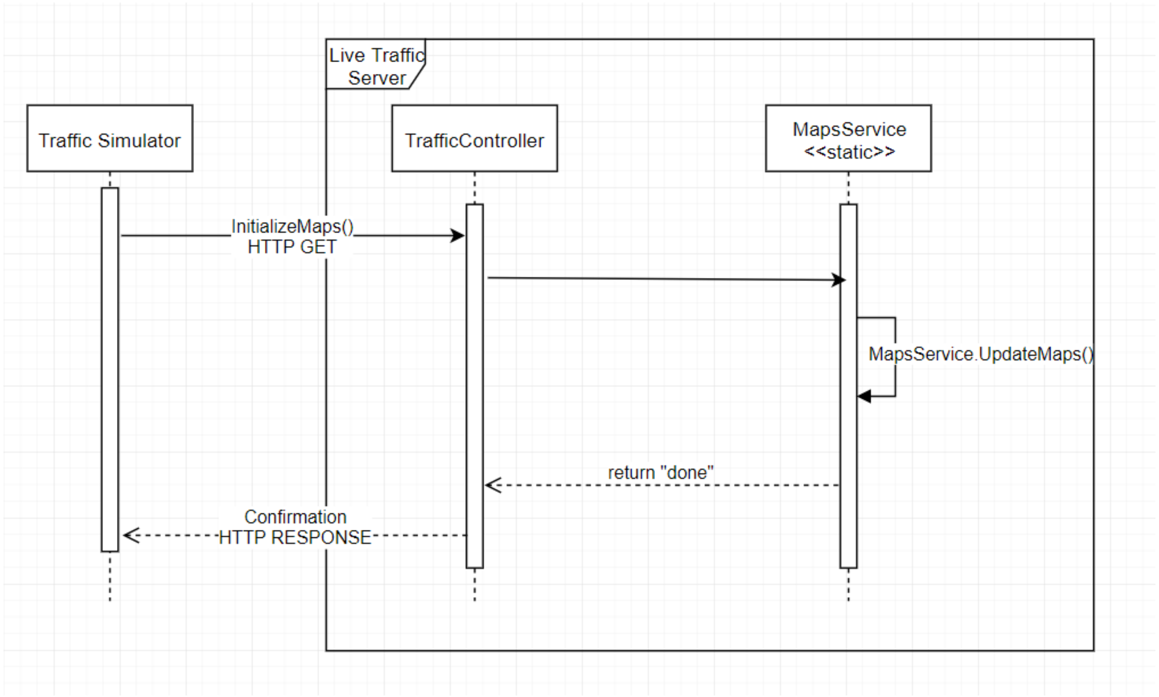


Figure 5.9 Flow Diagram of the InitializeMaps use-case in the Live Traffic Server

In Figure 5.9 the code flow of the InitializeMaps use-case is described. As stated earlier, the command is received via HTTP GET by the TrafficController. The controller delivers the responsability to the MapsService which handles the command as described at point 5.2.1.

- **The GetRoute flow**

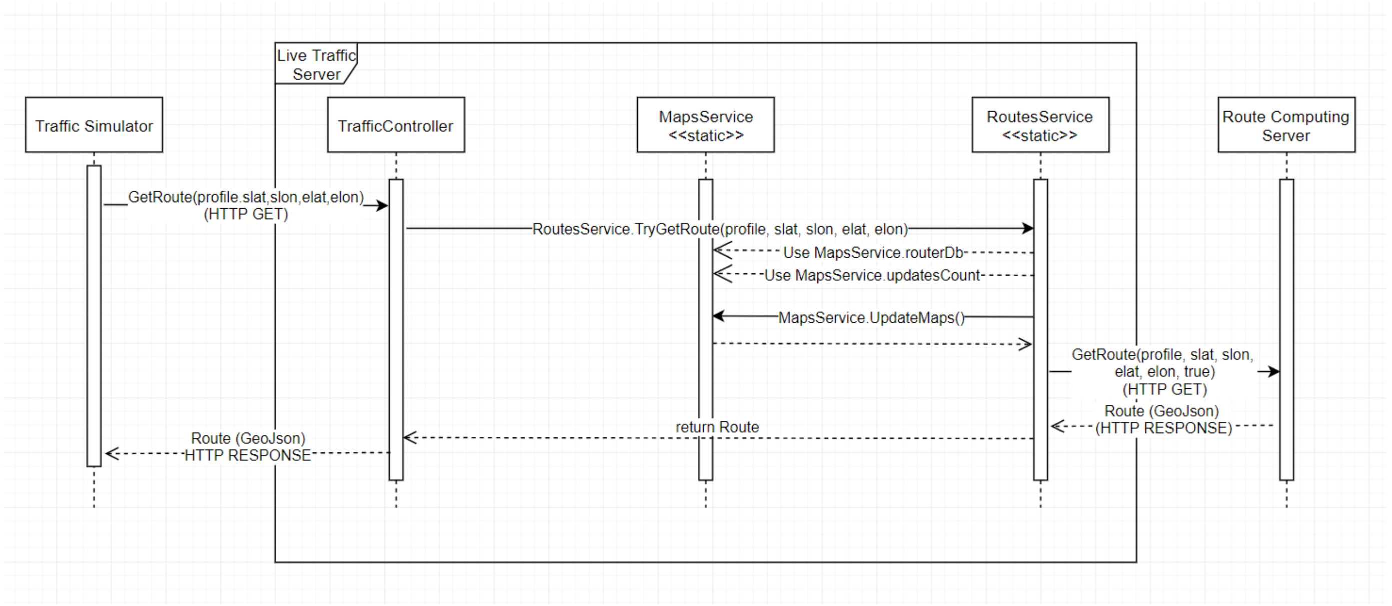

Live Traffic Server
Traffic Simulator
TrafficController
MapsService <<static>>
RoutesService <<static>>
Route Computing Server
GetRoute(profile.slat,slon,elat,elon) (HTTP GET)
RoutesService.TryGetRoute(profile, slat, slon, elat, elon)
Use MapsService.routerDb
Use MapsService.updatesCount
MapsService.UpdateMaps()
GetRoute(profile, slat, slon, elat, elon, true) (HTTP GET)
Route (GeoJson) (HTTP RESPONSE)
return Route
Route (GeoJson) HTTP RESPONSE

Figure 5.10 Flow diagram of the GetRoute use-case in the Live Traffic Server

In Figure 5.10 the code flow of the GetRoute use-case is described. As stated in 5.2.1, the command is received by the TrafficController via HTTP protocol. The controller delivers responsability to the RoutesService by using the TryGetRoute method. The service uses the routerDb and updatesCount variables from the MapsService and, if necesarry, it updates the routerDb file using the UpdateMaps method in the MapsService. The server then calls the Route Computing server via HTTP GET and returns the response received from it to the controller.

- **The UpdateLocation flow**

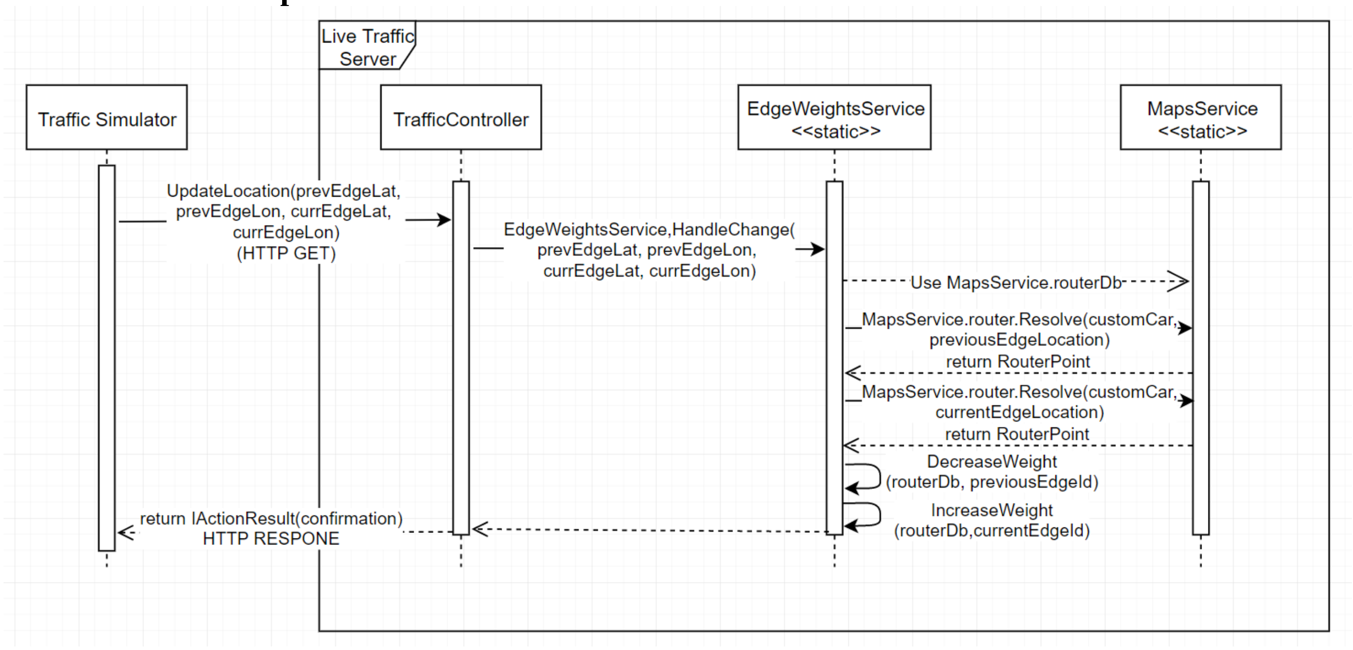

Live Traffic Server
Traffic Simulator
TrafficController
EdgeWeightsService <<static>>
MapsService <<static>>
UpdateLocation(prevEdgeLat, prevEdgeLon, currEdgeLat, currEdgeLon) (HTTP GET)
EdgeWeightsService.HandleChange(prevEdgeLat, prevEdgeLon, currEdgeLat, currEdgeLon)
Use MapsService.routerDb
MapsService.router.Resolve(customCar, previousEdgeLocation)
return RouterPoint
MapsService.router.Resolve(customCar, currentEdgeLocation)
return RouterPoint
DecreaseWeight (routerDb, previousEdgeId)
IncreaseWeight (routerDb,currentEdgeId)
return IActionResult(confirmation) HTTP RESPONE

Figure 5.11 Flow diagram of UpdateLocation use-case in the Live Traffic Server

In Figure 5.11 the code flow of the UpdateLocation use-case is described. As stated earlier, the command is received via HTTP GET by the TrafficController. The controller delivers the responsability to the EdgeWeightsService, by calling he HandleChange method. Using the routerDb and the router from the MapsService, the EdgeWeightsService snaps the input coordinates to the map, decreases the weight of the previous edge and increases the weight of the current edge.

5.3. Route Computing Server

5.3.1. Design

The Route Computing Server is a .NET Core 2.1 API that, as described at point 4.2.2, is the component that is responsible with managing routing requests from the Live Traffic Server.

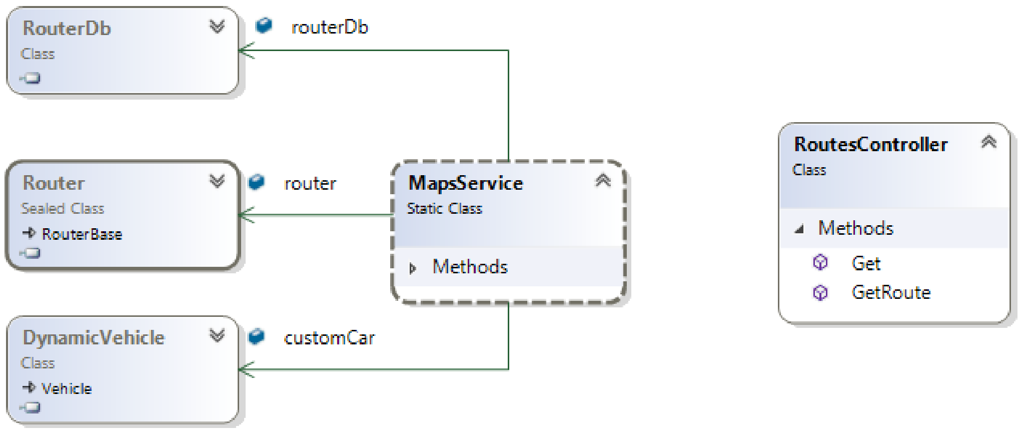


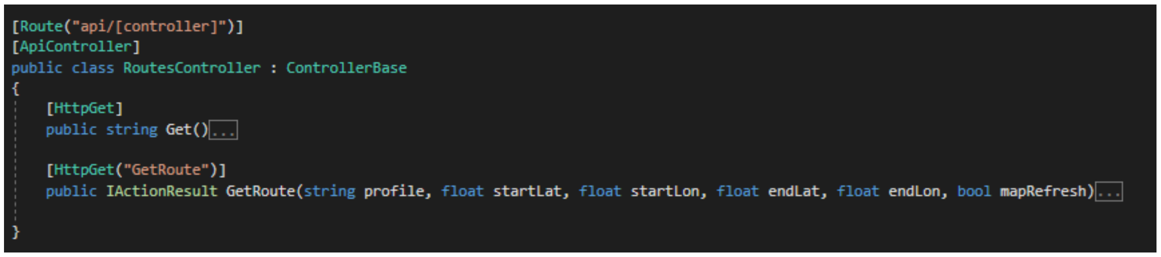
Figure 5.12 High level class diagram of Route Computing Server (from Visual Studio)

In Figure 5.12 the Route Computing Server is presented from a high level perspective. This diagram does not include the class structure behind the RouterDb class, as this diagram presents only the API project. That makes the high level class diagram rather simple.

The server contains 2 main classes: the controller called RoutesController and the static service that handles the maps, called MapsService.

- **RoutesController**

  The RoutesController class is an extension of the ControllerBase class in .NET Core. It is responsible with exposing the endpoints and handling the requests. There are two endpoints exposed by the RoutesController.



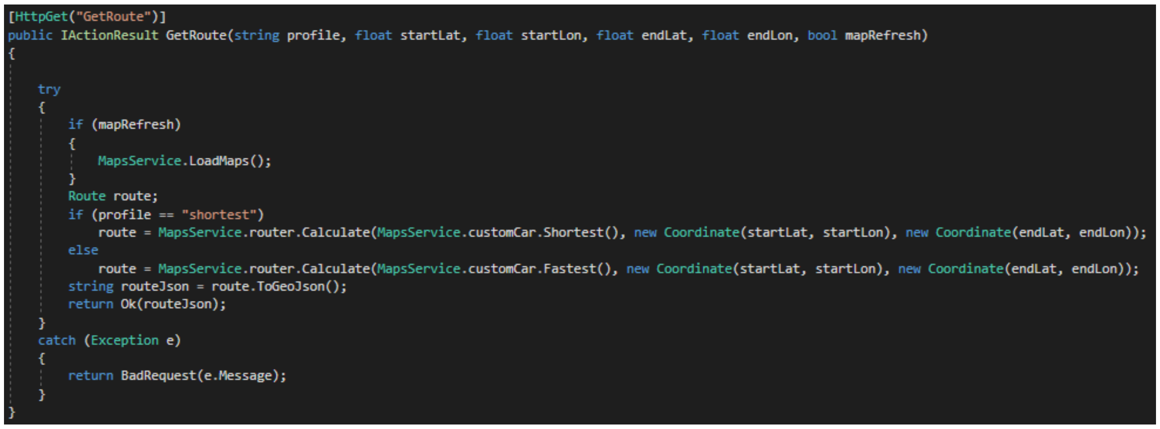
```
[Route("api/[controller]")]
[ApiController]
public class RoutesController : ControllerBase
{
    [HttpGet]
    public string Get()...

    [HttpGet("GetRoute")]
    public IActionResult GetRoute(string profile, float startLat, float startLon, float endLat, float endLon, bool mapRefresh)...

}
```

Figure 5.13 Screenshot of RoutesController endpoints code, colapsed to definitions

  Figure 5.13 presents the deffinitions of the endpoints in RoutesController as a screenshot from Visual Studio.

  The first endpoint, Get(), is a test endpoint. It returns the message "Route Computing Server is Running. Waiting for commands." if the server is running.

  The second endpoint, GetRoute, is responsible with receiving and handling route requests. It receives as parameters a routing profile, “shortest” or “best”, four float values that represent the coordinates of the start and end locations of the requested route and a flag announcing the server whether it should refresh its current map status or it should route using the currently loaded one.



```
[HttpGet("GetRoute")]
public IActionResult GetRoute(string profile, float startLat, float startLon, float endLat, float endLon, bool mapRefresh)
{

    try
    {
        if (mapRefresh)
        {
            MapsService.LoadMaps();
        }
        Route route;
        if (profile == "shortest")
            route = MapsService.router.Calculate(MapsService.customCar.Shortest(), new Coordinate(startLat, startLon), new Coordinate(endLat, endLon));
        else
            route = MapsService.router.Calculate(MapsService.customCar.Fastest(), new Coordinate(startLat, startLon), new Coordinate(endLat, endLon));
        string routeJson = route.ToGeoJson();
        return Ok(routeJson);
    }
    catch (Exception e)
    {
        return BadRequest(e.Message);
    }
}
```

Figure 5.14 Screenshot of the code in the GetRoute endpoint

Figure 5.14 presents the code of the GetRoute andpoint in a screenshot from Visual Studio.

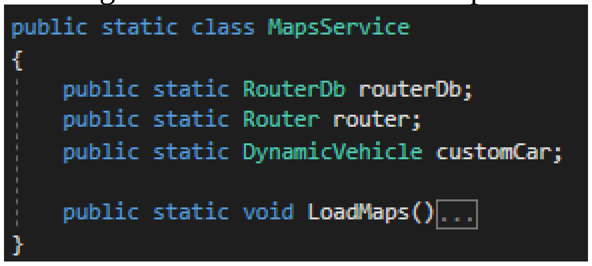
The first thing done is checking wether the mapRefresh flag is active or not. If it is, the LoadMaps method from the MapsService is called, in order to reload the maps.

The second step is computing the route. Based on the chosen routing profile, the controller calls the Calculate method of the router member in the MapsService and sends as parameters the routing profile chosen by the caller, and the two coordinates for the starting and ending point of the route. It then serializes the received route into a GeaoJson format end returns it to the caller with status OK.

All these steps are surrounded with a try/catch clause, so in case any of the steps fail, the error message is returned to the caller with status BadRequest.

- **MapsService**

  The Maps Service is a static service that is responsible with loading and maintaining the current status of the maps.



```
public static class MapsService
{
    public static RouterDb routerDb;
    public static Router router;
    public static DynamicVehicle customCar;

    public static void LoadMaps()...
}
```

Figure 5.15 Screenshot of the members and method of the MapsService

Figure 5.15 presents the members and the method definition of the Maps service as screenshot from Visual studio. The Maps service has three static members and a static method:

- **routerDb static member**

  The routerDb static member, of type RouterDb (from Itinero TAG), is the member that holds all the map and traffic information. It is loaded at the start of the server and is updated constantly using the LoadMaps method.

  - **router static member**
    The router static member, of type Router (from Itinero TAG), is the member that is used for routing in the routerDb. It is reinstantiated every time the maps are updated in the LoadMaps static method.
  - **customCar static member**
    The custom car static member, of type DynamicVehicle (from Itinero TAG), is the member that defines the profiles of the routings done in the routerDb. This is loaded from a custom Lua script.
  - **LoadMaps static method**
    The LoadMaps static method is responsible with refreshing the current routerDb by loading the new sattus from the routerDb file.
- Other functionalities
  All the functionalities concerning snaping and routing are implemented in Itinero and are described in TAG.

5.3.2. Flow

The Route Computing Server has one main flow: the GetRoute flow. This flow Starts when a route request is received by the RoutesController. An overview of the flow can be seen in Figure 5.16.

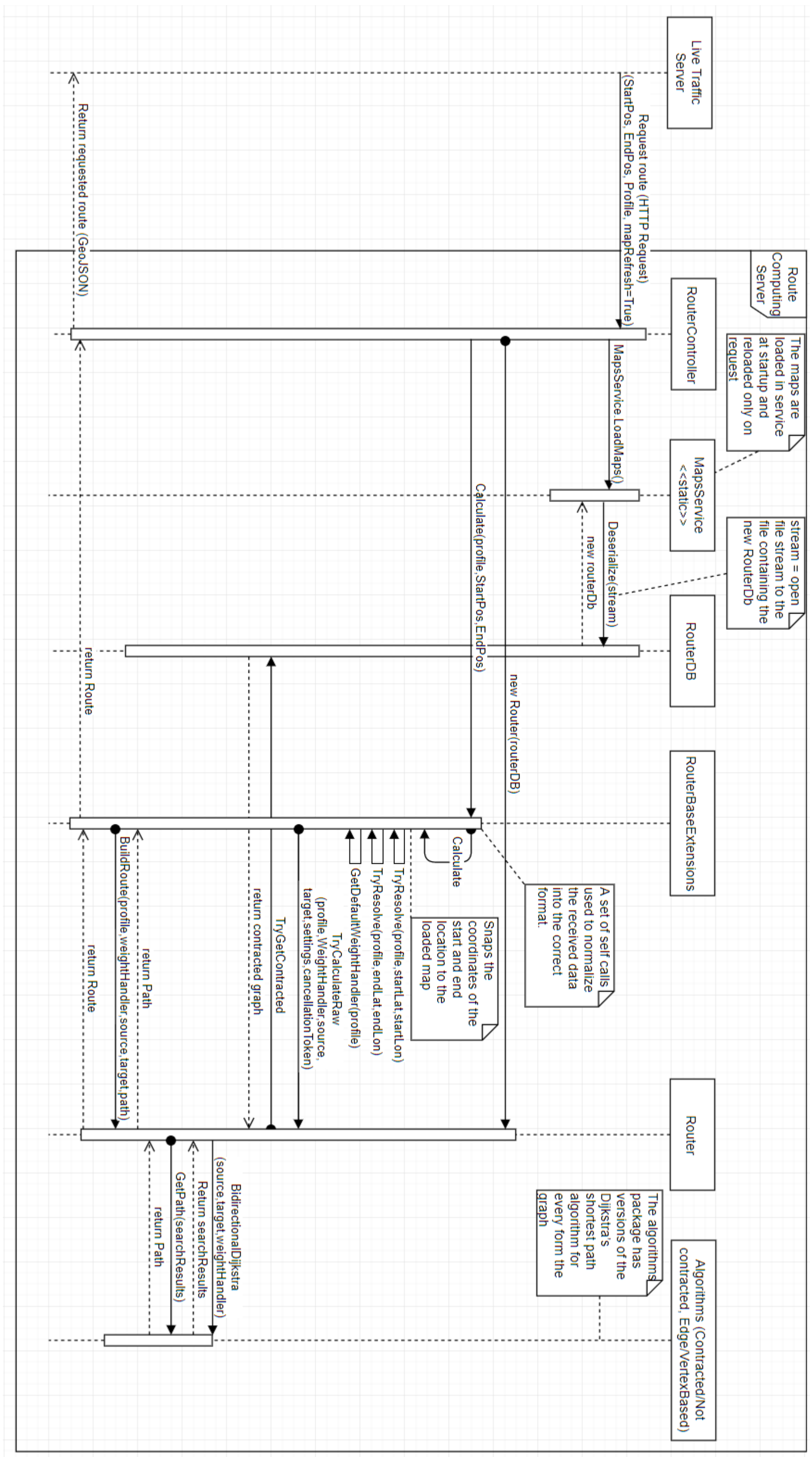

Live Traffic Server
Route Computing Server
RouterController
MapsService <<static>>
RouterDB
RouterBaseExtensions
Router
Algorithms (Contracted/Not contracted, Edge/VertexBased)
Request route (HTTP Request) (StartPos, EndPos, Profile, mapRefresh=True)
Return requested route (GeoJSON)
The maps are loaded in service at startup and reloaded only on request
MapsService.LoadMaps()
stream = open file stream to the file containing the new RouterDb
Deserialize(stream)
new routerDb
new Router(routerDB)
Calculate(profile,StartPos,EndPos)
return Route
A set of self calls used to normalize the received data into the correct format.
Calculate
Snaps the coordinates of the start and end location to the loaded map
TryResolve(profile,startLat,startLon)
TryResolve(profile,endLat,endLon)
GetDefaultWeightHandler(profile)
TryCalculateRaw (profile,WeightHandler,source, target,settings.cancellationToken)
TryGetContracted
return contracted graph
return Path
BuildRoute(profile,weightHandler,source,target,path)
return Route
The algorithms package has versions of the Djikstra's shortest path algorithm for every form the graph
BidirectionalDijkstra (source,target,weightHandler)
Return searchResults
GetPath(searchResults)
return Path

Figure 5.16 Flow diagram of a route request with map refresh

Figure 5.16 is a flow diagram of the code path in the Route Computing Server for a route request. In the use case described the mapRefresh flag in the request is set to true, so the maps have to be reloaded from the routerDb file before starting the route computation.

5.4. Traffic Simulator

5.4.1. Design

The Traffic Simulator is a .NET Framework 4.6.1 Console Application that is responsible with creating a virtual traffic context. The simulator creates virtual Traffic Participants that mimic the actions real Traffic Participants: they request routes, update their positions and ask for reroutes periodically.

The Traffic Simulator also gathers statistics from the traffic participants and processes them in order to output System and Simulation Statistics (6.4).

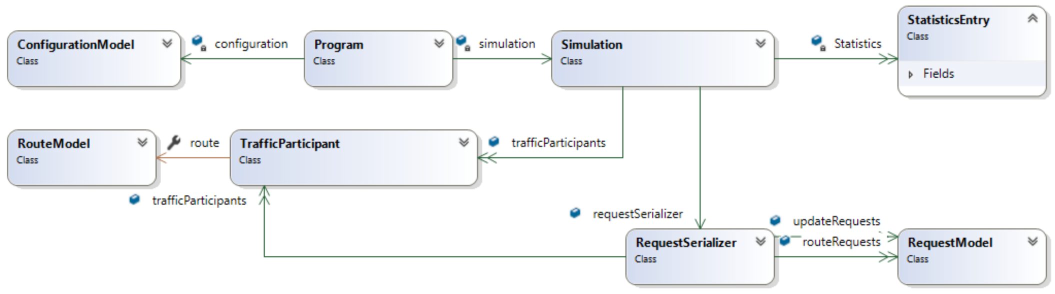
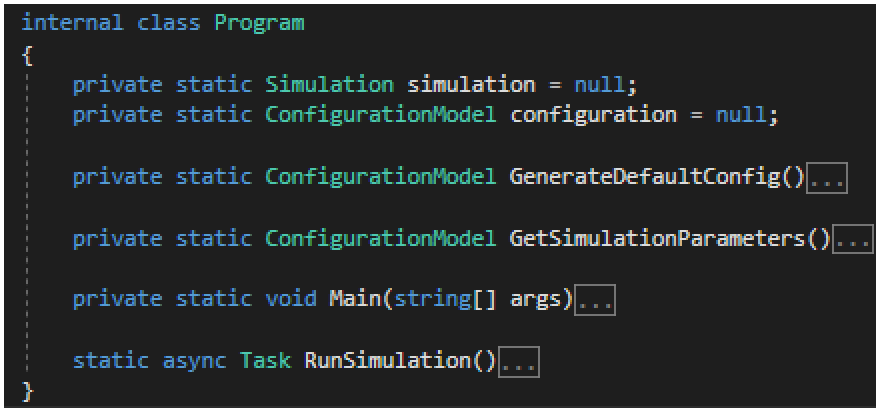


Figure 5.17 High level class diagram of Traffic Simulator (from Visual Studio)

In Figure 5.17, the Traffic simulator is described using a high level class diagram. The simulator is composed of four main classes and uses 4 types of models.

- **Program class**

  The Program class is the first class that is loaded when the simulator is started.



```
internal class Program
{
    private static Simulation simulation = null;
    private static ConfigurationModel configuration = null;

    private static ConfigurationModel GenerateDefaultConfig()...

    private static ConfigurationModel GetSimulationParameters()...

    private static void Main(string[] args)...

    static async Task RunSimulation()...
}
```

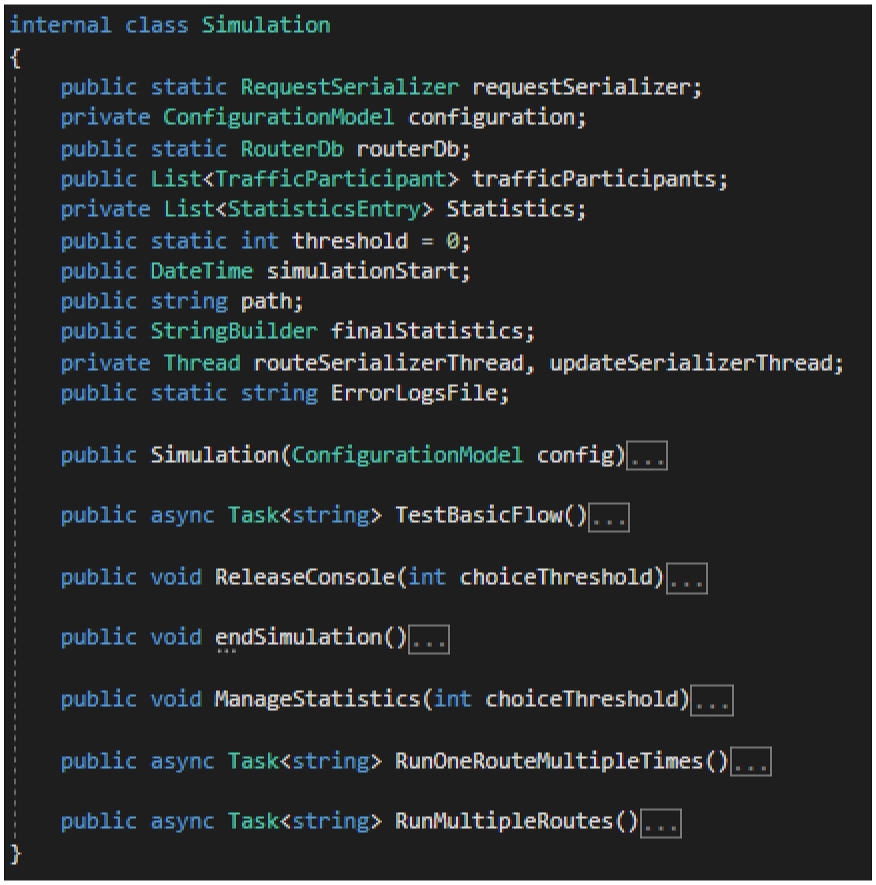
Figure 5.18 Screenshot of the structure of the Program class, methods colapsed to definitions

In Figure 5.18 the code of the Program class is presented usign a screenshot from Visual Studio. The class contains two static members and three static methods.

- simulation – instance of the Simulation class that represents a simulation with the parameters fond in the configuration
- configuration – instance of the ConfigurationModel class that holds all the information about the curently run simulation
- GenerateDefaulConfigs – method that generates a basic configurations file
- GetSimulationParameters – method that reads the simulation parameters from the SimulationParameters.json file and popultes the configuration field
- Main – method that is run when the project is built
- SunSimulation – method that instantiates the simulation parameter and runs the proper kind of simulation based on the simulation parameters

- **Simulation class**

  The Simulation class contains all the members and methods necessary for a simulation. It also contains separate methods that define the flow for every type of simulation.



```
internal class Simulation
{
    public static RequestSerializer requestSerializer;
    private ConfigurationModel configuration;
    public static RouterDb routerDb;
    public List<TrafficParticipant> trafficParticipants;
    private List<StatisticsEntry> Statistics;
    public static int threshold = 0;
    public DateTime simulationStart;
    public string path;
    public StringBuilder finalStatistics;
    private Thread routeSerializerThread, updateSerializerThread;
    public static string ErrorLogsFile;

    public Simulation(ConfigurationModel config)...

    public async Task<string> TestBasicFlow()...

    public void ReleaseConsole(int choiceThreshold)...

    public void endSimulation()...

    public void ManageStatistics(int choiceThreshold)...

    public async Task<string> RunOneRouteMultipleTimes()...

    public async Task<string> RunMultipleRoutes()...
}
```

Figure 5.19 Screenshot of the code in the Simulation class, methodes collapsed to definitions

- **TrafficParticipant class**
- **RequestSerializer class**

5.4.2. Flow

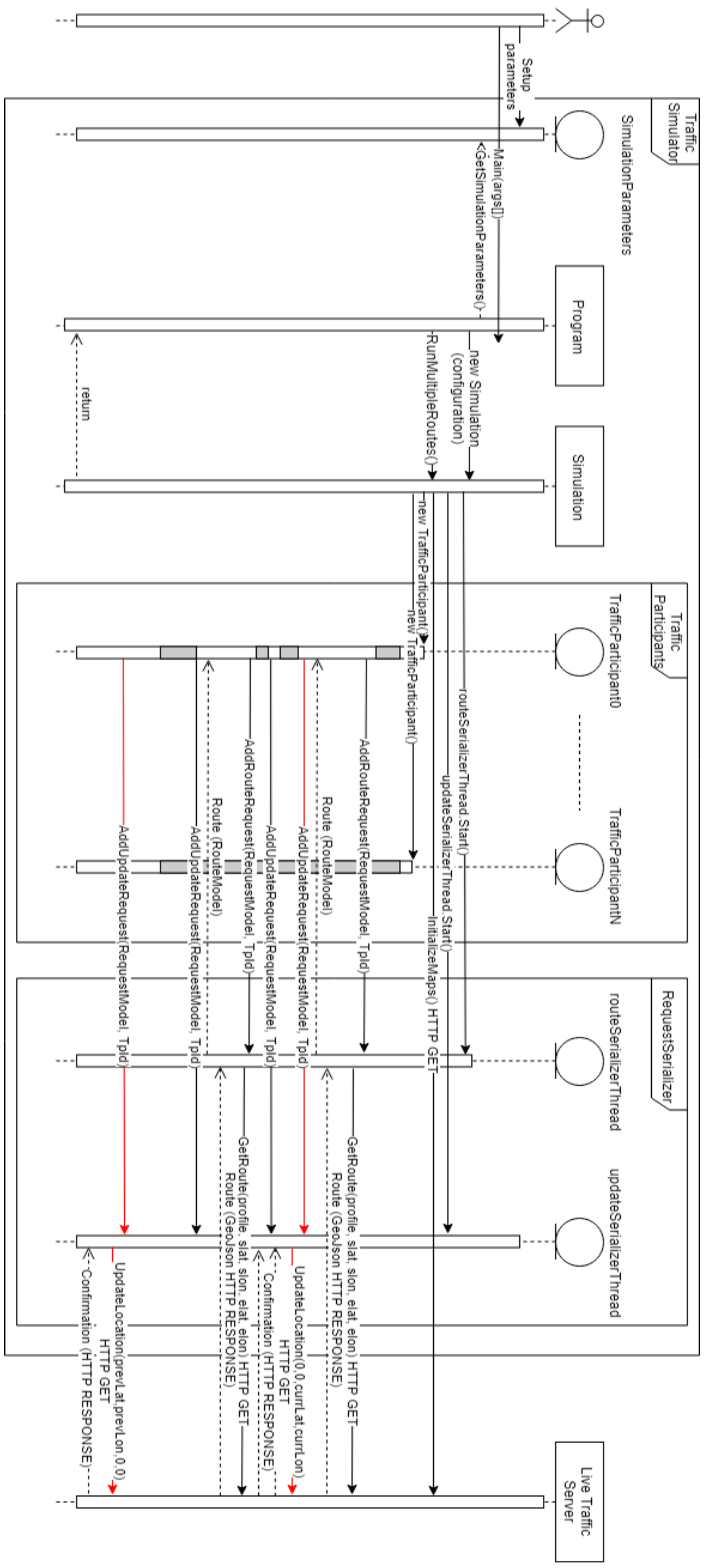

Traffic Simulator
SimulationParameters
Program
Simulation
Traffic Participants
TrafficParticipant0
TrafficParticipantN
RequestSerializer
routeSerializerThread
updateSerializerThread
Live Traffic Server
Setup parameters
Main(args[])
<GetSimulationParameters()
new Simulation (configuration)
RunMultipleRoutes()
return
new TrafficParticipant()
new TrafficParticipant()
routeSerializerThread.Start()
updateSerializerThread.Start()
InitializeMaps() HTTP GET
AddRouteRequest(RequestModel, Tpid)
Route (RouteModel)
AddUpdateRequest(RequestModel, Tpid)
AddUpdateRequest(RequestModel, Tpid)
AddRouteRequest(RequestModel, Tpid)
Route (RouteModel)
AddUpdateRequest(RequestModel, Tpid)
AddUpdateRequest(RequestModel, Tpid)
GetRoute(profile, slat, slon, elat, elon) HTTP GET
Route (GeoJson HTTP RESPONSE)
UpdateLocation(0,0,currLat,currLon) HTTP GET
Confirmation (HTTP RESPONSE)
GetRoute(profile, slat, slon, elat, elon) HTTP GET
Route (GeoJson HTTP RESPONSE)
UpdateLocation(prevLat,prevLon,0,0) HTTP GET
Confirmation (HTTP RESPONSE)

Figure 5.20 Sequence Diagram of the Traffic Simulator code

Chapter 6. Testing and Validation

6.1. Hardware

Testing was done using 3 different setups:

1. Intel Core i5-4460 3.20 GHz, 16 GB RAM, 100 GB free SSD storage
2. Intel Core i7-4720 2.60 GHz, 16 GB RAM, 60 GB free SSD storage
3. Intel Core i5-6700 2.60 GHz, 16 GB RAM, 100 GB free SSD storage

6.2. Metrics

In order to study the efficiency of the presented project, metrics should be defined for both the system itself and the effects of the system in the connected cars context, computed using the implemented Traffic Simulator (4.3.1.).

6.2.1. System metrics

The system metrics are the characteristics of the system, measuring the way this works. These metrics would show the capabilities of the system in any module configuration. They focus especially on the Live Traffic Server and on the Route Computing Server.

- Live Traffic Server route request response time: The time it takes for the Live Traffic Server to respond to a request for a route.
- Route Computing Server route request response time: The time it takes for the Route Computing Server to receive a route request, compute that route and respond to the request.
- Live Traffic Server Update request response time: The time it takes for the Live Traffic Server to respond to an Update Request.

6.2.2. Simulation metrics

The Simulation metrics are the characteristics of the results obtained by running different scenarios in the Traffic Simulator. These metrics are used to analyse and compare different approaches of implementation or different traffic scenarios.

- Simulation length: the amount of time it took for the Traffic Simulator to finish the job
- Total time: the sum of the durations of all routes simulated
- Average route time: the average duration of the routes simulated
- Average speed: the average speed of the Traffic Participants simulated
- Average touched features: the average number of edge sets touched by the Traffic participants while following the simulated routes
- Average route request wait time: the average time the simulated traffic participants had to wait for one route request
- Average update request wait time: the average time the simulated traffic participants had to wait for one update request

6.3. Scenario

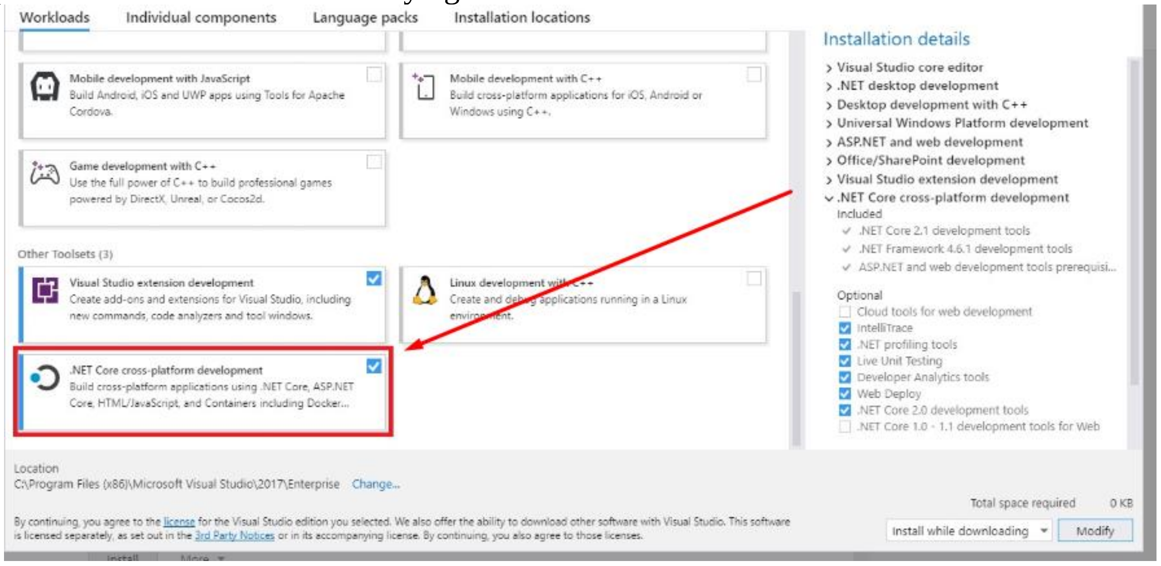
6.4. Results

Chapter 7. User’s manual

This chapter comprises the installation of all the components of the developed system as well as methods to test each one of these components. Furthermore, all the development and testing were done on only one machine at a time, with some supplementary steps the system can be easily distributed. The choice of testing only on one machine at a time was made so the speed of the simulation was as big as possible.

7.1. Environment

For running of the current version of the system the user needs the Visual Studio IDE. The development was done using Microsoft Visual Studio Express 2017. Using a different version of Visual Studio might create issues. .NET Core 2.0 was used in the development of the Live Traffic Server and Route Computing server so the user must make sure the .NET Core SDK is installed, by checking the “.NET Core cross-platform development” in the installation/modifying wizard of Visual Studio.



Mobile development with JavaScript [ ]
Build Android, iOS and UWP apps using Tools for Apache Cordova.

Mobile development with C++ [ ]
Build cross-platform applications for iOS, Android or Windows using C++.

Game development with C++ [ ]
Use the full power of C++ to build professional games powered by DirectX, Unreal, or Cocos2d.

Other Toolsets (3)

Visual Studio extension development [✓]
Create add-ons and extensions for Visual Studio, including new commands, code analyzers and tool windows.

Linux development with C++ [ ]
Create and debug applications running in a Linux environment.

.NET Core cross-platform development [✓]
Build cross-platform applications using .NET Core, ASP.NET Core, HTML/JavaScript, and Containers including Docker...

Installation details
- Visual Studio core editor
- .NET desktop development
- Desktop development with C++
- Universal Windows Platform development
- ASP.NET and web development
- Office/SharePoint development
- Visual Studio extension development
- .NET Core cross-platform development
  - Included
    - ✓ .NET Core 2.1 development tools
    - ✓ .NET Framework 4.6.1 development tools
    - ✓ ASP.NET and web development tools prerequisi...
  - Optional
    - [ ] Cloud tools for web development
    - [✓] IntelliTrace
    - [✓] .NET profiling tools
    - [✓] Live Unit Testing
    - [✓] Developer Analytics tools
    - [✓] Web Deploy
    - [✓] .NET Core 2.0 development tools
    - [ ] .NET Core 1.0 - 1.1 development tools for Web

Location
C:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise Change...

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

Total space required 0 KB

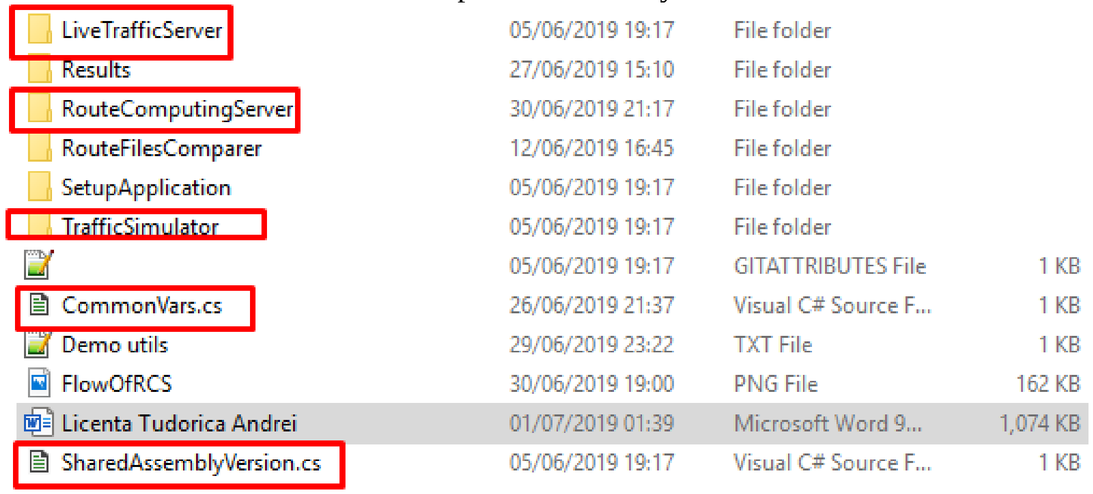
Figure 7.1 Visual Studio installation wizzard

7.2. Dependencies

Each module of the system uses a few external elements. These elements have to be present in order for the system to work. In the case of a distribution of the system over multiple machines the following steps must be done for each machine.

- Download the project from [5]
- Create a folder destinated for map related files
- Create a folder destinated for the results of the simulation
- Download the map of the country of focus from [6]
- Reduce the map to the specific area of focus using instructions from [7] (the smaller the area, the faster the system)
- Add the .pbf file of the area to the folder created at step 1.

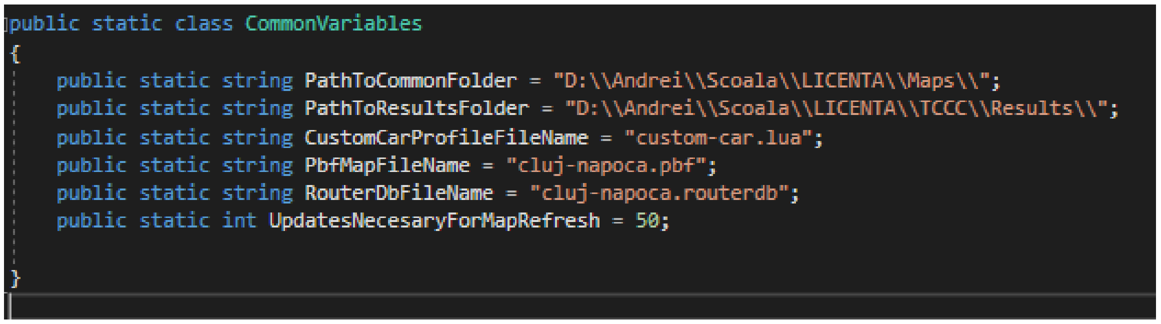
- Make sure that the CommonVars.cs and SharedAssemblyVersion.cs files are in the same folder with the components of the system on the machine.



LiveTrafficServer	05/06/2019 19:17	File folder	
Results	27/06/2019 15:10	File folder	
RouteComputingServer	30/06/2019 21:17	File folder	
RouteFilesComparer	12/06/2019 16:45	File folder	
SetupApplication	05/06/2019 19:17	File folder	
TrafficSimulator	05/06/2019 19:17	File folder	
	05/06/2019 19:17	GITATTRIBUTES File	1 KB
CommonVars.cs	26/06/2019 21:37	Visual C# Source F...	1 KB
Demo utils	29/06/2019 23:22	TXT File	1 KB
FlowOfRCS	30/06/2019 19:00	PNG File	162 KB
Licenta Tudorica Andrei	01/07/2019 01:39	Microsoft Word 9...	1,074 KB
SharedAssemblyVersion.cs	05/06/2019 19:17	Visual C# Source F...	1 KB

Figure 7.2 Screenshot of the folder containing components and common files

- Open the CommonVars.cs file
- Change the value of "PathToCommonFolder" to be the path of the folder you created for the maps
- Change the value of "PathToResultsFolder" to be the path of the folder you created for the results
- Add the custom-car.lua file, that can be found in the root folder of the system, in the folder destinated for maps
- Change the "PbfFileName" value to be equal to the name of the final map to be used
- Change the "RouterDbFileName" to match with the target area
- Run the SetupApplication found in the same folder with all the components

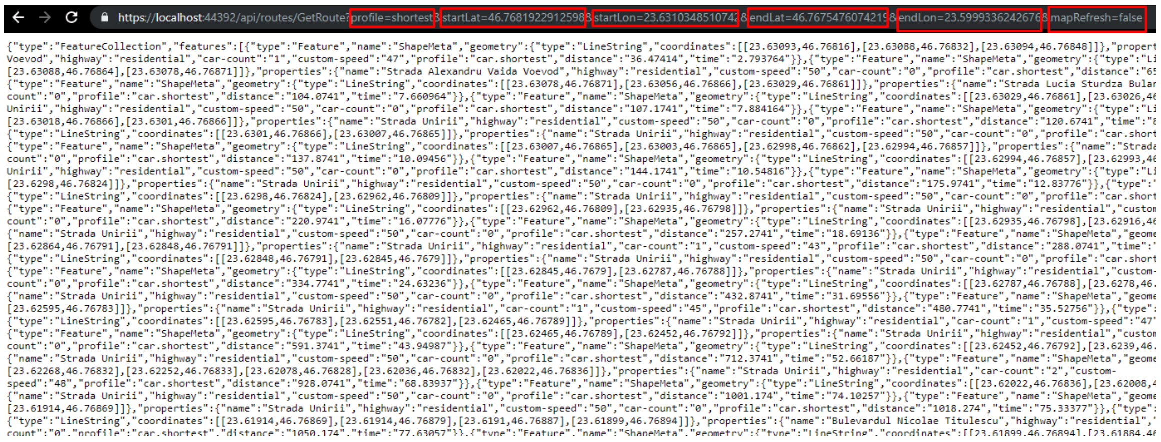


```
public static class CommonVariables
{
    public static string PathToCommonFolder = "D:\\Andrei\\Scoala\\LICENTA\\Maps\\";
    public static string PathToResultsFolder = "D:\\Andrei\\Scoala\\LICENTA\\TCCC\\Results\\";
    public static string CustomCarProfileFileName = "custom-car.lua";
    public static string PbfMapFileName = "cluj-napoca.pbf";
    public static string RouterDbFileName = "cluj-napoca.routerdb";
    public static int UpdatesNecesaryForMapRefresh = 50;

}
```

Figure 7.3 Screenshot of CommonVars.cs in current version

7.3. Route Computing Server

The Route Computing Server code can be found in the "RouteComputingServer" folder of the downloaded system. After all the steps at 7.2. are completed, run the .sln file and compile the project. This should open a page saying the Route Computing Server is running and waiting for commands. Testing can be done by building a link containing 2 locations on the chosen map.

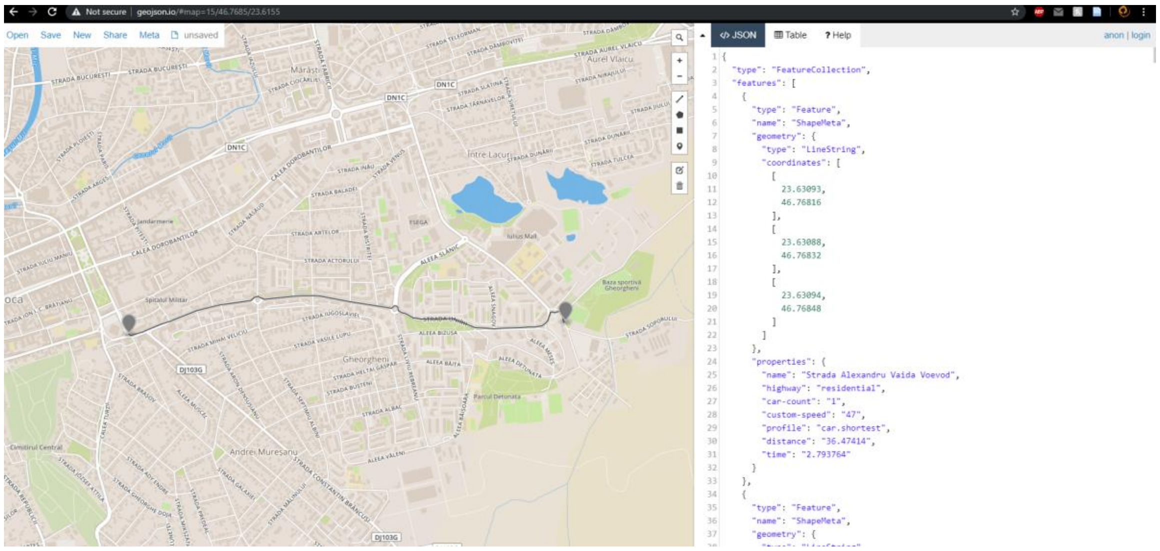


```
https://localhost:44392/api/routes/GetRoute?profile=shortest&startLat=46.7681922912598&startLon=23.6310348510742&endLat=46.7675476074219&endLon=23.5999336242676&mapRefresh=false
```

```
{"type":"FeatureCollection","features":[{"type":"Feature","name":"ShapeMeta","geometry":{"type":"LineString","coordinates":[[23.63093,46.76816],[23.63088,46.76832],[23.63094,46.76848]]},"propert
Voevod","highway":"residential","car-count":"1","custom-speed":"47","profile":"car.shortest","distance":"36.47414","time":"2.793764"}},{"type":"Feature","name":"ShapeMeta","geometry":{"type":"Li
[23.63088,46.76864],[23.63078,46.76871]]},"properties":{"name":"Strada Alexandru Vaida Voevod","highway":"residential","custom-speed":"50","car-count":"0","profile":"car.shortest","distance":"65
{"type":"Feature","name":"ShapeMeta","geometry":{"type":"LineString","coordinates":[[23.63078,46.76871],[23.63056,46.76866],[23.63029,46.76861]]},"properties":{"name":"Strada Lucia Sturdza Bular
count":"0","profile":"car.shortest","distance":"104.0741","time":"7.660964"}},{"type":"Feature","name":"ShapeMeta","geometry":{"type":"LineString","coordinates":[[23.63029,46.76861],[23.63026,46
Unirii","highway":"residential","custom-speed":"50","car-count":"0","profile":"car.shortest","distance":"107.1741","time":"7.884164"}},{"type":"Feature","name":"ShapeMeta","geometry":{"type":"Li
[23.63018,46.76866],[23.6301,46.76866]]},"properties":{"name":"Strada Unirii","highway":"residential","custom-speed":"50","car-count":"0","profile":"car.shortest","distance":"120.6741","time":"8
{"type":"LineString","coordinates":[[23.6301,46.76866],[23.63007,46.76865]]},"properties":{"name":"Strada Unirii","highway":"residential","custom-speed":"50","car-count":"0","profile":"car.short
{"type":"Feature","name":"ShapeMeta","geometry":{"type":"LineString","coordinates":[[23.63007,46.76865],[23.63003,46.76865],[23.62998,46.76862],[23.62994,46.76857]]},"properties":{"name":"Strada
count":"0","profile":"car.shortest","distance":"137.8741","time":"10.09456"}},{"type":"Feature","name":"ShapeMeta","geometry":{"type":"LineString","coordinates":[[23.62994,46.76857],[23.62993,46
Unirii","highway":"residential","custom-speed":"50","car-count":"0","profile":"car.shortest","distance":"144.1741","time":"10.54816"}},{"type":"Feature","name":"ShapeMeta","geometry":{"type":"Li
[23.6298,46.76824]]},"properties":{"name":"Strada Unirii","highway":"residential","custom-speed":"50","car-count":"0","profile":"car.shortest","distance":"175.9741","time":"12.83776"}},{"type":"
{"type":"LineString","coordinates":[[23.6298,46.76824],[23.62962,46.76809]]},"properties":{"name":"Strada Unirii","highway":"residential","custom-speed":"50","car-count":"0","profile":"car.short
{"type":"Feature","name":"ShapeMeta","geometry":{"type":"LineString","coordinates":[[23.62962,46.76809],[23.62935,46.76798]]},"properties":{"name":"Strada Unirii","highway":"residential","custom
count":"0","profile":"car.shortest","distance":"220.9741","time":"16.07776"}},{"type":"Feature","name":"ShapeMeta","geometry":{"type":"LineString","coordinates":[[23.62935,46.76798],[23.62916,46
{"name":"Strada Unirii","highway":"residential","custom-speed":"50","car-count":"0","profile":"car.shortest","distance":"257.2741","time":"18.69136"}},{"type":"Feature","name":"ShapeMeta","geome
[23.62864,46.76791],[23.62848,46.76791]]},"properties":{"name":"Strada Unirii","highway":"residential","car-count":"1","custom-speed":"43","profile":"car.shortest","distance":"288.0741","time":"
{"type":"LineString","coordinates":[[23.62848,46.76791],[23.62845,46.7679]]},"properties":{"name":"Strada Unirii","highway":"residential","custom-speed":"50","car-count":"0","profile":"car.short
{"type":"Feature","name":"ShapeMeta","geometry":{"type":"LineString","coordinates":[[23.62845,46.7679],[23.62787,46.76788]]},"properties":{"name":"Strada Unirii","highway":"residential","custom-
count":"0","profile":"car.shortest","distance":"334.7741","time":"24.63236"}},{"type":"Feature","name":"ShapeMeta","geometry":{"type":"LineString","coordinates":[[23.62787,46.76788],[23.6278,46.
{"name":"Strada Unirii","highway":"residential","custom-speed":"50","car-count":"0","profile":"car.shortest","distance":"432.8741","time":"31.69556"}},{"type":"Feature","name":"ShapeMeta","geome
[23.62595,46.76783]]},"properties":{"name":"Strada Unirii","highway":"residential","car-count":"1","custom-speed":"45","profile":"car.shortest","distance":"480.7741","time":"35.52756"}},{"type":
{"type":"LineString","coordinates":[[23.62595,46.76783],[23.62551,46.76782],[23.62465,46.76789]]},"properties":{"name":"Strada Unirii","highway":"residential","car-count":"1","custom-speed":"47"
{"type":"Feature","name":"ShapeMeta","geometry":{"type":"LineString","coordinates":[[23.62465,46.76789],[23.62452,46.76792]]},"properties":{"name":"Strada Unirii","highway":"residential","custom
count":"0","profile":"car.shortest","distance":"591.3741","time":"43.94987"}},{"type":"Feature","name":"ShapeMeta","geometry":{"type":"LineString","coordinates":[[23.62452,46.76792],[23.6239,46.
{"name":"Strada Unirii","highway":"residential","custom-speed":"50","car-count":"0","profile":"car.shortest","distance":"712.3741","time":"52.66187"}},{"type":"Feature","name":"ShapeMeta","geome
[23.62268,46.76832],[23.62252,46.76833],[23.62078,46.76828],[23.62036,46.76832],[23.62022,46.76836]]},"properties":{"name":"Strada Unirii","highway":"residential","car-count":"2","custom-
speed":"48","profile":"car.shortest","distance":"928.0741","time":"68.83937"}},{"type":"Feature","name":"ShapeMeta","geometry":{"type":"LineString","coordinates":[[23.62022,46.76836],[23.62008,4
{"name":"Strada Unirii","highway":"residential","custom-speed":"50","car-count":"0","profile":"car.shortest","distance":"1001.174","time":"74.10257"}},{"type":"Feature","name":"ShapeMeta","geome
[23.61914,46.76869]]},"properties":{"name":"Strada Unirii","highway":"residential","custom-speed":"50","car-count":"0","profile":"car.shortest","distance":"1018.274","time":"75.33377"}},{"type":
{"type":"LineString","coordinates":[[23.61914,46.76869],[23.61914,46.76879],[23.6191,46.76887],[23.61899,46.76894]]},"properties":{"name":"Bulevardul Nicolae Titulescu","highway":"residential","
count":"0","profile":"car.shortest","distance":"1050.174","time":"77.63057"}},{"type":"Feature","name":"ShapeMeta","geometry":{"type":"LineString","coordinates":[[23.61899,46.76894],[23.61884,46
```

Figure 7.4 Screenshot of test run for the Route Computing Server

Figure 7.4. presents an example of usage for the following link: https://localhost:44392/api/routes/GetRoute?profile=shortest&startLat=46.7681922912598&startLon=23.6310348510742&endLat=46.7675476074219&endLon=23.5999336242676&mapRefresh=false. The response of the Route Computing Server is a GeoJson string containing a route. By pasting this into [8] the visualization of this route is obtained.


Not secure | geojson.io/#map=15/46.7685/23.6155
Open Save New Share Meta unsaved
JSON Table ? Help
anon | login

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "name": "ShapeMeta",
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [
            23.63093,
            46.76816
          ],
          [
            23.63088,
            46.76832
          ],
          [
            23.63094,
            46.76848
          ]
        ]
      },
      "properties": {
        "name": "Strada Alexandru Vaida Voevod",
        "highway": "residential",
        "car-count": "1",
        "custom-speed": "47",
        "profile": "car.shortest",
        "distance": "36.47414",
        "time": "2.793764"
      }
    },
    {
      "type": "Feature",
      "name": "ShapeMeta",
      "geometry": {
```

Figure 7.5 Screenshot of the visualization of the response received by the command in Figure 7.4.

For best performance the Route Computing server should be done in Release mode. This can be done by selecting „Release” in the run mode dropdown menu and pressing CTRL+F5.

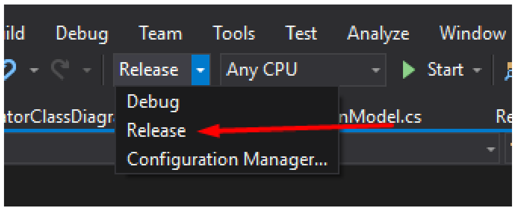


Figure 7.6 screenshot of run mode dropdown menu in Visula Studio

7.4. Live Traffic Server

The Live Traffic Server code can be found in the “LiveTrafficServer” folder of the downloaded system. After all the steps at 7.2. are completed, open the .sln file. Open the Constants.cs file in Visual Studio and modify the value of “RouteComputingServerURL” so that it matches the URL where the Route Computing Server. In a local environment this is "https://localhost:44392/".

When the project is run a page saying “Live Traffic Server is Running. Waiting for commands.” should open.

The server initializes the maps by itself at start up. The functionality can be tested by building a link containing 2 locations on the chosen map just like the one for the Route Computing server.

Example:

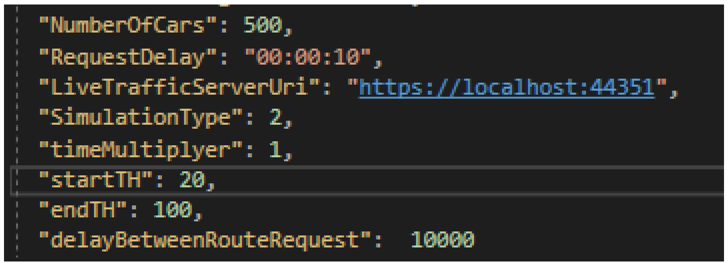
https://localhost:44351/api/traffic/GetRoute?profile=shortest&startLat=46.7681922912598&startLon=23.6310348510742&endLat=46.7675476074219&endLon=23.5999336242676

The visualization of the answer follows the same steps ast the ones for the Route Computing Server.

For best performance the Route Computing server should be done in Release mode. This can be done by selecting „Release” in the run mode dropdown menu (Figure 7.6) and pressing CTRL+F5.

7.5. Traffic Simulation

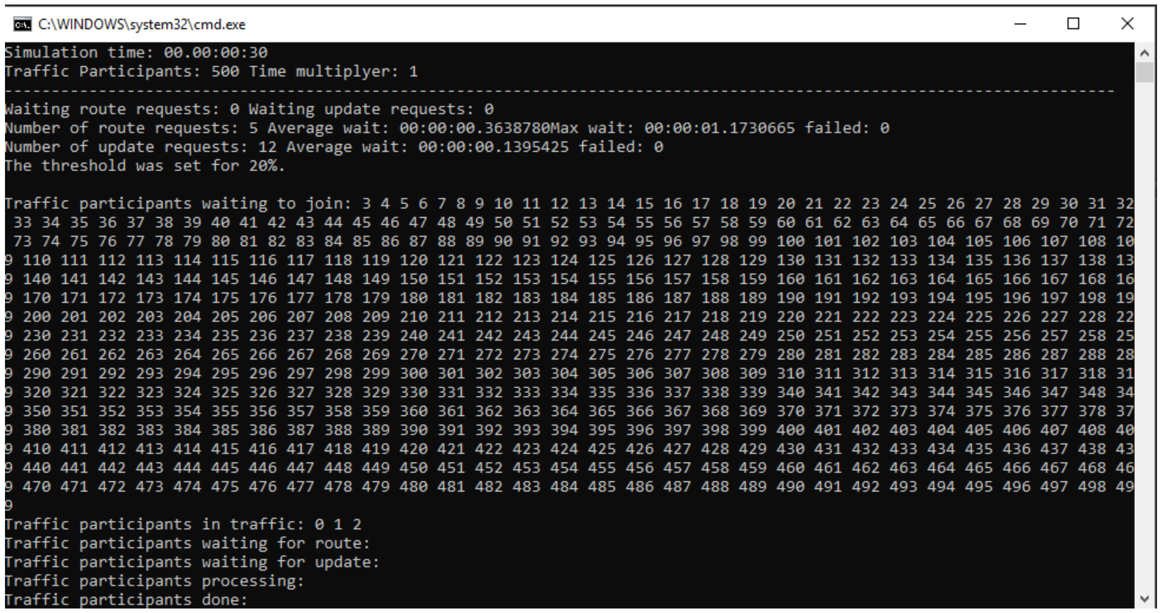
The Traffic Simulator code can be found in the “TrafficSimulator” folder. After all the steps at 7.2. are completed, open the .sln file. Open the SimulationParameters.json file in Visual Studio and modify the value of the “LiveTrafficServerUri” such that it matches the URL where the running Live Traffic Server can be found. In a local environment that link should be "https://localhost:44351". Modify the other simulation parameters so they match the desired simulation.



```
"NumberOfCars": 500,
"RequestDelay": "00:00:10",
"LiveTrafficServerUri": "https://localhost:44351",
"SimulationType": 2,
"timeMultiplyer": 1,
"startTH": 20,
"endTH": 100,
"delayBetweenRouteRequest":  10000
```

Figure 7.7 Example of simulations parameters.

Run the simulator in release mode to get live statistics about the simulation.

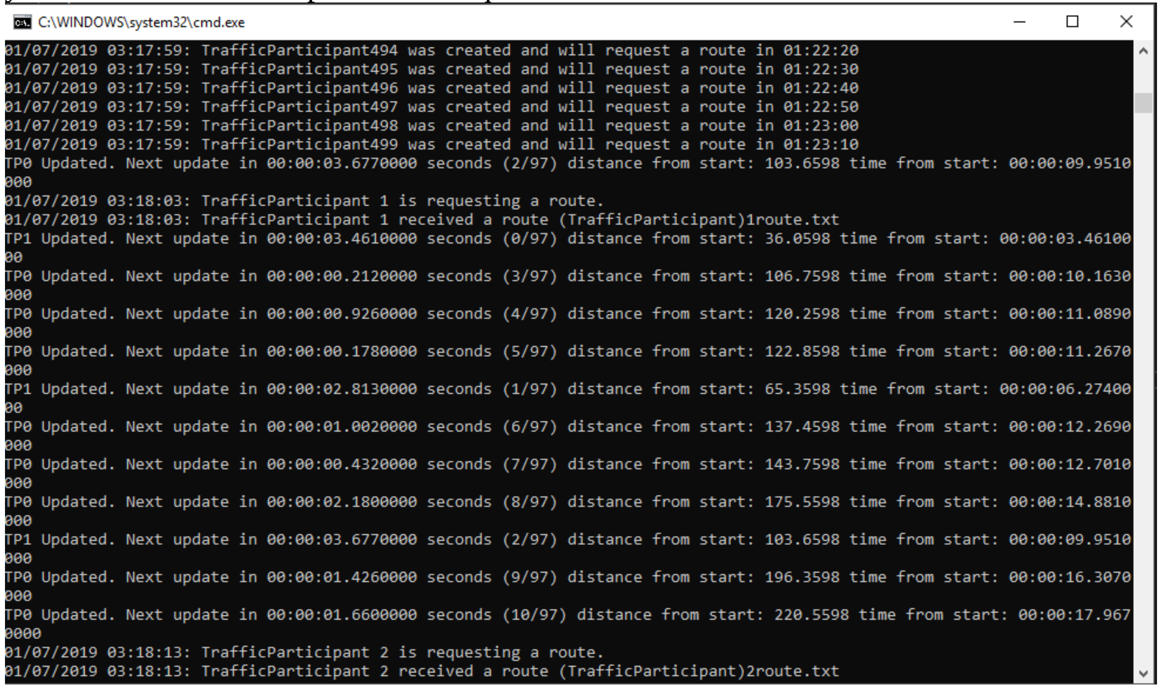


C:\WINDOWS\system32\cmd.exe

```
Simulation time: 00.00:00:30
Traffic Participants: 500 Time multiplyer: 1
-------------------------------------------------------------------------------------------------------------
Waiting route requests: 0 Waiting update requests: 0
Number of route requests: 5 Average wait: 00:00:00.3638780Max wait: 00:00:01.1730665 failed: 0
Number of update requests: 12 Average wait: 00:00:00.1395425 failed: 0
The threshold was set for 20%.

Traffic participants waiting to join: 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 10
9 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 13
9 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 16
9 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 19
9 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 22
9 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 25
9 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 28
9 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 31
9 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 34
9 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 37
9 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 40
9 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 43
9 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 46
9 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 49
9
Traffic participants in traffic: 0 1 2
Traffic participants waiting for route:
Traffic participants waiting for update:
Traffic participants processing:
Traffic participants done:
```

Figure 7.8 Screenshot of simulator run in Release mode

Run the simulator in Debug mode to follow the Simulation step by step. This might be really hard to followas the updates move quite fast.



C:\WINDOWS\system32\cmd.exe

```
01/07/2019 03:17:59: TrafficParticipant494 was created and will request a route in 01:22:20
01/07/2019 03:17:59: TrafficParticipant495 was created and will request a route in 01:22:30
01/07/2019 03:17:59: TrafficParticipant496 was created and will request a route in 01:22:40
01/07/2019 03:17:59: TrafficParticipant497 was created and will request a route in 01:22:50
01/07/2019 03:17:59: TrafficParticipant498 was created and will request a route in 01:23:00
01/07/2019 03:17:59: TrafficParticipant499 was created and will request a route in 01:23:10
TP0 Updated. Next update in 00:00:03.6770000 seconds (2/97) distance from start: 103.6598 time from start: 00:00:09.9510
000
01/07/2019 03:18:03: TrafficParticipant 1 is requesting a route.
01/07/2019 03:18:03: TrafficParticipant 1 received a route (TrafficParticipant)1route.txt
TP1 Updated. Next update in 00:00:03.4610000 seconds (0/97) distance from start: 36.0598 time from start: 00:00:03.46100
00
TP0 Updated. Next update in 00:00:00.2120000 seconds (3/97) distance from start: 106.7598 time from start: 00:00:10.1630
000
TP0 Updated. Next update in 00:00:00.9260000 seconds (4/97) distance from start: 120.2598 time from start: 00:00:11.0890
000
TP0 Updated. Next update in 00:00:00.1780000 seconds (5/97) distance from start: 122.8598 time from start: 00:00:11.2670
000
TP1 Updated. Next update in 00:00:02.8130000 seconds (1/97) distance from start: 65.3598 time from start: 00:00:06.27400
00
TP0 Updated. Next update in 00:00:01.0020000 seconds (6/97) distance from start: 137.4598 time from start: 00:00:12.2690
000
TP0 Updated. Next update in 00:00:00.4320000 seconds (7/97) distance from start: 143.7598 time from start: 00:00:12.7010
000
TP0 Updated. Next update in 00:00:02.1800000 seconds (8/97) distance from start: 175.5598 time from start: 00:00:14.8810
000
TP1 Updated. Next update in 00:00:03.6770000 seconds (2/97) distance from start: 103.6598 time from start: 00:00:09.9510
000
TP0 Updated. Next update in 00:00:01.4260000 seconds (9/97) distance from start: 196.3598 time from start: 00:00:16.3070
000
TP0 Updated. Next update in 00:00:01.6600000 seconds (10/97) distance from start: 220.5598 time from start: 00:00:17.967
0000
01/07/2019 03:18:13: TrafficParticipant 2 is requesting a route.
01/07/2019 03:18:13: TrafficParticipant 2 received a route (TrafficParticipant)2route.txt
```

Figure 7.9 screenshot of simulator run in Debug mode

The results of the simulation can be found on the console during the simulation and at the end of it. Also, at the end of the simulation, in the Results folder that was created at 7.2, the simulator creates files for the entire simulation statistics and for each traffic participant.

For best performance the Simulator should be done in Release mode.

7.6. Distribution

In order for the system to be distributed a couple of changes should be made.

First of all, the live traffic server's code should be modified so it writes the routerDb in the route request to the Route Computing Server, not in a local file. Also, the Route Computings code shoud be modified such that the Route Computing receives a routerDb in the route request body and deserializes it in the local RouterDb structure.

The second part would be changing the IIS settings and making the server discoverable, just like in [9]. This can be done for both the Live Traffic Server and for the Route Computing Server.

Chapter 8. Conclusions

About. 5% of the whole.
In this chapter you present:

- A summary of your contributions/achievements,
- A critical analysis of the results achieved,
- A description of the possibilities of improvements/further development.

Bibliography

[1] “OsmAnd Web page,” [Online]. Available: https://osmand.net/.

[2] “OsmAnd Repository,” [Online]. Available: https://github.com/osmandapp/Osmand.

[3] “Itinero Web Page,” [Online]. Available: https://www.itinero.tech/.

[4] B. Abelshausen, “Linked In page,” Itinero, [Online]. Available: https://www.linkedin.com/in/benabelshausen/?originalSubdomain=be.

[5] A. Tudorica, “Licence Thesis Repo,” [Online]. Available: https://github.com/andreitudorica/UTCN_LICENTA.

[6] “OpenStreetMap Data Extracts,” [Online]. Available: https://download.geofabrik.de/.

[7] “OsmConvert Wiki,” [Online]. Available: https://wiki.openstreetmap.org/wiki/Osmconvert.

[8] “GeoJson,” [Online]. Available: http://geojson.io.

[9] “Remote Access to Local ASP.NET Core Applications,” [Online]. Available: https://blog.kloud.com.au/2017/02/27/remote-access-to-local-aspnet-core-apps-from-mobile-devices/.

Appendix 1 (only if needed)

…
Relevant code sections
…
Other relevant info (proofs etc.)
…
Published papers (if any)
etc.